

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



趣链科技创始人及核心团队撰写

区块链
技术丛书

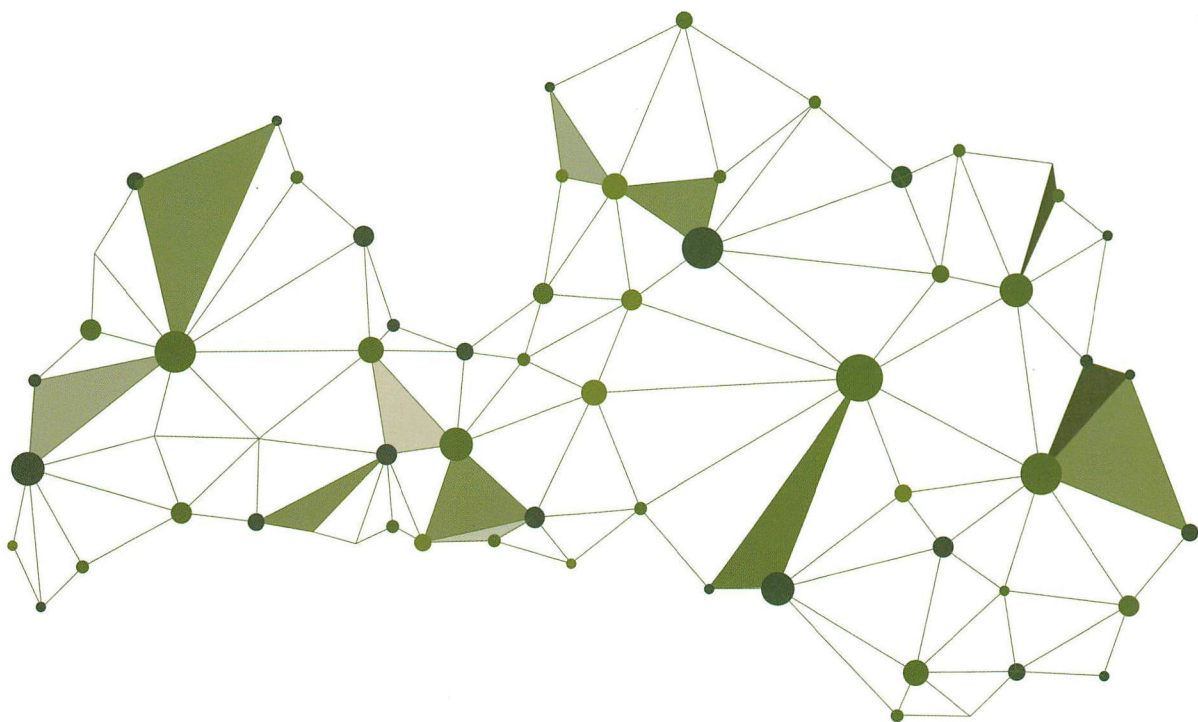
从源代码角度剖析Hyperledger Fabric项目的整体架构设计，
以及各核心模块的实现原理

HZ BOOKS
华章IT

THE SOURCE CODE ANALYSIS
OF HYPERLEDGER FABRIC

Hyperledger Fabric 源代码分析与深入解读

蔡亮 梁秀波 宣章炯 ◎ 著



机械工业出版社
China Machine Press





容简介

这是一本深度解读 Hyperledger Fabric 架构设计与实现原理的著作，由国内知名区块链公司趣链科技的创始人和核心技术团队成员撰写。

全书的核心内容以 Hyperledger Fabric 的源代码为切入点，首先从宏观上分析了 Hyperledger Fabric 项目的整体架构与设计，然后深入源代码详细分析了 Hyperledger Fabric 各个重要模块的设计与实现原理。此外，为了兼顾没有区块链开发基础的读者，书中还加入了 Hyperledger Fabric 开发环境搭建、综合案例、项目部署等实战性内容，可使读者能在深入理解 Hyperledger Fabric 设计机制的基础上快速动手实践。

全书共 14 章，逻辑上分为两大部分

第一部分：源码分析（第 2~11 章）

第 2 章首先从宏观的角度解读 Hyperledger Fabric 的整体架构、项目的结构，以及交易流程，这为后面的源码分析打下基础；

第 3 章分析了 Logging 日志模块、Error 错误处理框架、Config 配置模块、GRPC 服务 4 个模块的源码，对理解后续的源码有帮助；

第 4~11 章深入纤细地分析了 Peer、Order、Chaincode、MSP、Gossip、BCCSP、Fabric-CA、账本机制等节点和功能的设计与实现，这部分内容能让读者全面、透彻了解整个 Hyperledger Fabric 的运作机制。

第二部分：开发实战（第 1 章及第 12~14 章）

第 1 章主要是为开发 Hyperledger Fabric 应用做准备，讲解了 Go 语言开发环境的准备，以及 Docker 环境的准备；

第 12~14 章分别讲解了一个智能合约的案例、完整的 Hyperledger Fabric 项目案例，以及项目的部署方法。

实战部分不仅能提升读者的动手实践能力，而且还能辅助他们更好地理解源码分析的内容，使理论和实践完美融合到一起。





华章IT
HZBOOKS | Information Technology







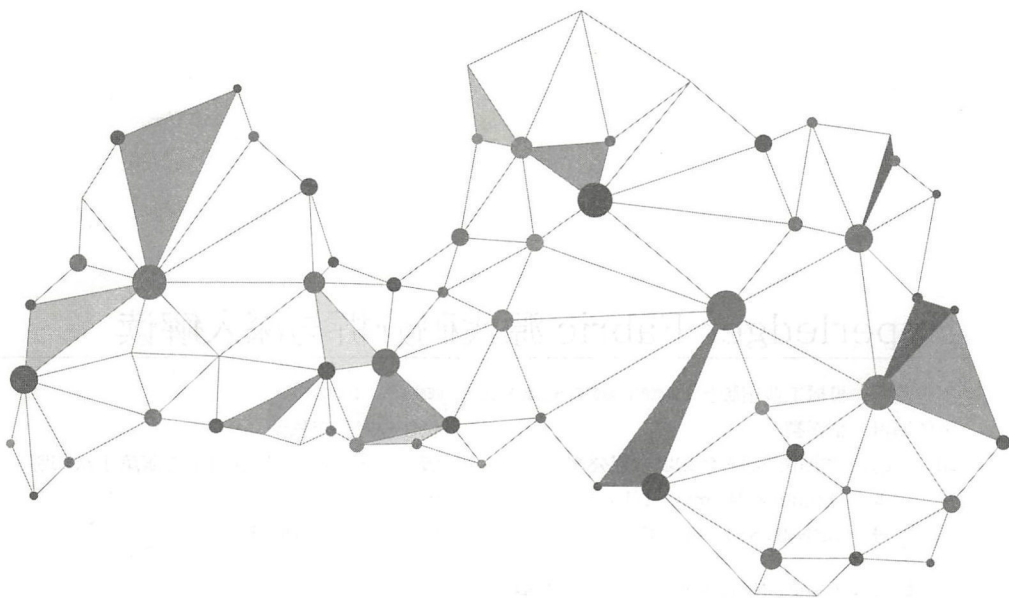
区块链
技术丛书

THE SOURCE CODE ANALYSIS
OF HYPERLEDGER FABRIC

Hyperledger Fabric

源代码分析与深入解读

蔡亮 梁秀波 宣章炯 © 著



机械工业出版社
China Machine Press





图书在版编目 (CIP) 数据

Hyperledger Fabric 源代码分析与深入解读 / 蔡亮, 梁秀波, 宣章炯著. —北京: 机械工业出版社, 2018.9
(区块链技术丛书)

ISBN 978-7-111-60870-7

I. H… II. ①蔡… ②梁… ③宣… III. 电子商务 - 支付方式 - 研究 IV. F713.361.3

中国版本图书馆 CIP 数据核字 (2018) 第 210324 号

Hyperledger Fabric 源代码分析与深入解读

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 张锡鹏

责任校对: 李秋荣

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2018 年 9 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 25

书 号: ISBN 978-7-111-60870-7

定 价: 89.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东



Foreword 序 —

以企业级联盟区块链技术引领我国新一代信息技术的突破式发展

区块链技术及其理念被认为是近年来最具颠覆性和革命性的创新之一，近年来已在金融、贸易、物流、征信、公益、物联网、共享经济等诸多领域崭露头角，被世界各国高度重视。美国、俄罗斯、英国、德国等国相继把区块链上升为国家战略。未来，区块链技术及其理念将对社会生活的各个不同领域产生重要影响，凡是涉及数据共享、信息可信和价值传递的场景都离不开区块链技术的应用。可以说，区块链技术及其理念作为信息共享和信任协作的基础，将重构人类社会未来的生产关系，通过与人工智能、云计算、大数据和物联网等新技术新理念的结合，将极大地提高社会生产力。

区块链将成为未来数字化社会的底层基础设施，从底层技术层面上为各行各业的持续稳定发展提供信任支撑，其价值主要体现在这些方面：（1）减少中间环节，实现实时清算结算，通过智能合约提高业务的自动化程度，从而为各行各业带来业务效率的提升；（2）实现数据的共享与标准化对接，基于数字资产的多行业流通实现价值网络的自繁殖，从而降低业务的拓展成本；（3）基于历史数据不可篡改、全流程可追溯和智能合约的规则约束可增强监管机构的监管能力；（4）该技术的多方可信合作和公平激励机制将成为未来分布式商业的发展基石，为各行业创造更多的合作机会。

在区块链技术快速发展的近十年中，产生了三次重要的技术演进：（1）以比特币为代表的加密数字货币应用将区块链技术引入了人们的视野；（2）以以太坊为代表的可编程区块链平台为区块链技术应用各个领域提供了可能；（3）以 Hyperledger Fabric 和 Hyperchain 为代表的企业级区块链平台使得区块链技术能够真正应用于商业环境。本书以 Hyperledger Fabric 为讲解对象，对其源代码进行了细致分析和深入解读，以期帮助读者更好地理解和使用企业级的区块链底层平台技术。

Hyperledger Fabric 是 Linux 基金会引导的开源联盟区块链项目，其核心目标是建立开放的、标准化的、企业级的、支持商业应用的分布式账本框架与基础代码，具有高效共识、智能合约、多级加密、权限控制、隐私保护等特性。相较于比特币和以太坊，Hyperledger





Fabric 的核心优势有：（1）实现模块化和可插拔，更能适应复杂的商业应用要求；（2）多通道特性保障数据安全性，不同通道之间实现数据隔离；（3）实现证书的强化管理，提供单独的 Fabric CA 项目；（4）具有较好的可扩展性，解耦了原子排序与其他复杂处理环节；（5）可根据负载进行灵活部署，将交易处理节点逻辑简化为背书节点和确认节点。

本书的撰写团队来自于浙江大学区块链研究中心和杭州趣链科技有限公司，具有丰富的区块链底层核心技术研发和上层商业应用开发的经验。由杭州趣链科技有限公司研发的国产自主可控的联盟区块链平台 Hyperchain 目前已在中国工商银行、中国农业银行、中国建设银行、中国银联、Google、美国道富银行、葡萄牙商业银行等数十家国内外大型公司和机构得到了落地应用，在业内产生了强烈反响。

本书对 Hyperledger Fabric 架构及其设计模式进行分析，深入解读其源代码，并结合 Fabric-sample 项目介绍相关实践技术。相信可以给区块链技术爱好者和相关行业从业人员提供参考，具有很好的理论和实践价值。

陈纯

中国工程院院士，浙江大学计算机科学与技术学院教授
曾任浙江大学软件学院院长和浙江大学计算机软件研究所所长





Foreword 序 二

去除炒作泡沫，回归技术本质

区块链的多中心、自动化和可信任的理念与特性使得它可成为未来信息化社会信息共享、业务协作、价值传输的底层技术支撑。近年来，随着区块链技术的飞速发展和区块链理念的探索应用，以金融、证券、保险、物联网、跨境贸易、能源等为代表的很多行业都开始应用区块链。近几年新注册成立的、业务与区块链有关的公司达数千家，区块链专利申请量由两年前的几百件猛增至五千余件，社会对区块链技术人才的需求非常大，浙江大学等知名高校率先开设了区块链课程，区块链技术和理念深入到社会生活的方方面面已成为大势所趋。随着人们对区块链技术和理念认识的深入，其去除泡沫、回归本质的趋势非常明显，人们讨论的热点逐渐从如何炒币赚钱转向了如何应用区块链来推动各领域的产业升级与变革，区块链的底层核心技术和上层应用探索成为了产业发展的重点。

区块链本质上是提供信任保障的技术，可能发展为未来互联网最重要的基础设施之一。若能构建以区块链技术为基础的价值传输网络，将深刻地改变未来的信息社会形态。以区块链为底层技术支撑的加密数字货币应用给传统金融体系带来了一定的挑战，各国政府和央行都对加密数字货币应用保持了密切的关注。在区块链技术去货币化的探索阶段，业界普遍认识到了该技术所蕴含的巨大价值，通过在区块链上构建可重用、模块化、自动化的智能合约，许多创新的应用场景不断涌现，例如征信管理、跨境贸易、资源共享和版权共享等。以 Hyperledger Fabric 和 Hyperchain 为代表的企业级区块链平台克服了传统区块链技术资源消耗严重、交易性能低下、缺乏隐私保护等问题，提供了图灵完备的智能合约支持，使得开发者可以更便捷地开发和部署各领域的区块链商业应用，大大促进了基于联盟链的区块链商业形态的蓬勃发展。

Hyperledger 是 Linux 基金会于 2015 年发起的推进区块链数字技术和交易验证的开源项目，目标是使得区块链技术在加密数字货币之外也可广泛应用于其他场景。项目成立之初，就有 IBM、摩根大通、思科、Intel 等科技和金融巨头加入。Fabric 是 Hyperledger 最重要的子项目之一，它通过支持插件组件的模块化架构，实现了完备的权限管理、创新



的一致性算法等相关工作，对区块链技术的发展产生了重要影响。对于国内众多想利用 Hyperledger Fabric 开发部署区块链应用的开发者而言，相关的中文专业资料过于稀少和浅显。本书对 Hyperledger Fabric 开发环境的搭建与部署进行了详细地介绍，并对其技术架构和源码进行了深入地分析，为相关开发人员提供了及时的技术参考。

本书的作者团队来自浙江大学区块链研究中心和杭州趣链科技有限公司，在区块链前沿技术研究和产业化应用方面积累丰富。浙江大学区块链研究中心是专注于区块链领域的浙江大学校级研究机构，其目标是攻克区块链的底层技术难关，有效推动我国自主、安全、可控区块链技术的快速发展和产业化推广。杭州趣链科技有限公司是近年来成长极为迅速的区块链平台和应用解决方案提供商，其研发的区块链底层平台 Hyperchian 已在众多大型商业公司和机构的生产环境中投入使用。深厚的理论功底和扎实的实践经验使得本书在理论分析和实践应用方面都颇具特色。相信本书可以在帮助读者深刻理解 Hyperledger Fabric 的技术内涵的基础上，有效提升自己在区块链方面的开发能力。

Julian Gordon

Hyperledger 亚太区副总裁



Preface 前言

区块链技术自诞生以来就受到社会各界的广泛关注，甚至被认为是继互联网之后最具颠覆性意义的技术革命。如果说最初的互联网实现了信息在全球范围内的高速传播与共享，那么区块链则致力于构建一个可用于价值交换的下一代可信互联网；如果说人工智能等技术改变的是生产力，那么区块链技术改变的是生产关系。据专家预测，区块链在不远的将来会引发政治、经济、文化、科技、军事、教育、医疗、能源等诸多领域的颠覆性革命。

区块链的本质是一个去中心化的分布式账本系统，是一种涵盖了分布式数据存储、点对点传输、共识机制、加密算法等一系列计算机技术的集成创新技术。区块链采用分布式数据库来存储数据，通过自信任的共识机制实现全网数据的一致性，将对称与非对称加密技术相结合以保证数据安全和不可篡改，采用智能合约技术规范数据化数据处理，最终实现在无需第三方中介机构的情况下进行人与人之间的价值交换，进而解决信任问题。

众所周知，区块链技术源于比特币，最初是作为比特币的底层技术而存在的。因此，区块链的发展也与比特币息息相关。2008年，一个化名为中本聪的人在一篇名为《比特币：一个点对点的电子现金系统》的论文中详细描述了一种去中心化的分布式账本系统，并把这个系统中所使用的数字加密货币称为比特币。比特币的诞生开启了区块链 1.0 时代，这一时期各种基于区块链技术的加密数字货币如雨后春笋般涌现出来，如以太坊、莱特币、瑞波币等。然而此时的区块链技术仅仅用于交易记账，应用范围非常有限。

随着各种加密数字货币的广泛流行，一些有识之士逐渐意识到区块链技术的巨大应用潜力。他们将区块链技术从比特币系统中分离出来，并通过添加智能合约技术构建出一个更通用的去中心化应用开发平台，其中最具代表性的就是以太坊。以太坊的出现标志着区块链迈入了 2.0 时代，这一时期区块链技术的应用范围得到了极大的扩展，同时涉及股权众筹、证券交易、贷款抵押等诸多金融领域。

然而，以太坊平台作为一个基于公有链的区块链平台，依然存在一些不足之处亟待改进，比如存在共识效率低下、隐私保护缺乏、大规模存储困难和信息难以监管等问题。为了应对大规模商业应用的业务需求，面向企业级应用的联盟区块链技术应运而生。其中的

典型代表就是 Hyperledger Fabric 开源项目，它通过模块化和可插拔设计、权限控制、多链和多通道等技术为区块链技术在各领域的应用打开了广阔的空间。

本书是一本介绍 Hyperledger Fabric 架构、源代码及其底层实现的专业书籍。从编程语言、源码分析、网络节点、加密算法、智能合约、架构设计等多个角度深入解读 Fabric 项目，并通过对项目实例的分析介绍，使读者能够理论结合实践，增强实际动手能力，更好地理解 Fabric 的实现原理。本书是一本实际操作性极强的 Hyperledger Fabric 的专业书籍，通过阅读本书，能够帮助读者零基础快速入门 Fabric。

本书结构

本书共分为 14 章。

第 1 章主要介绍如何从零开始，通过准备工作完成多种平台上 Hyperledger Fabric 应用环境的配置。首先，讲解不同系统中 Go 语言的安装与环境配置，并简单介绍了 Go 语言。之后，讲解不同系统中 Docker 的安装和简单使用方法。最后，介绍了 Hyperledger 社区与社区中一些常用软件，方便读者在完成环境配置之余，更好地了解 Hyperledger Fabric 的相关信息。

第 2 章主要讲解 Fabric 的架构并进行分析。首先介绍了 Fabric 整体架构，包括系统架构、交易背书及策略、证实账本和节点账本的检查。然后介绍了 Fabric 交易流程。接着介绍了 Fabric 整体项目结构，并详细阐述了 Fabric 源码中相关缩写的含义。

第 3 章主要介绍了 Fabric 中的四个模块：Logging 日志模块、Error 错误处理框架、Config 配置模块、grpc 服务。理解日志模块有助于理解源码；理解错误处理框架能使读者学会独立处理代码中常见错误的方法；理解配置模块对具象化程序很有帮助；理解 grpc 有助于读者理解如何实现客户端与服务器端的远程调用，从而为进一步理解源码打下基础。

第 4 章主要介绍了 Fabric 中 peer 节点的设计与实现。peer 节点作为 Fabric 中处理交易、存储区块的角色，承担了重要的作用。本章对 CommandLine 进行了解析，介绍了 Admin 及 Endorser 服务的实现。同时具体分析了 Committer 机制及其作用。

第 5 章主要介绍了 order 的设计与实现。orderer 为 Fabric 的排序节点，其为所有的客户端提供统一的交易排序及打包服务，并将区块分发到所有的 leader 节点。本章的内容包括 Orderer 内部机制窥探、kafka 排序服务机制讲解以及 orderer 在 Fabric 中的交互流程。

第 6 章主要介绍了 chaincode 的设计与实现。chaincode 是 Fabric 实现智能合约的方式，利用容器技术将智能合约放置在容器中运行，同时进行调用。本章的内容包括 chaincode 生命周期管理，原理浅析，数据结构分析，SystemChaincode 的讲解，CSCC、ESCC、LSCC、QSCC、VSCC 等概念的分析，SystemChaincode 的注册和实例化，ApplicationChaincode 的部署及实例化，以及 chaincode 操作步骤。

第7章主要介绍了MSP成员服务提供者及Fabric的交易网络。通过Membership Service Providers (MSP) 主题帮助读者更好地理解加密操作、签名、校验、认证等在Fabric中的应用。

第8章主要介绍了Gossip协议。Gossip协议是分布式系统用于保证分布式一致性的一种方式，也是Fabric中传递各种信息及区块信息的手段。本章后对Gossip协议原理进行了解析，介绍了Gossip的服务组件、服务初始化、消息广播、channel通道的设计与实现以及事件机制。

第9章主要介绍了Fabric中BCCSP加密服务提供者的设计与实现。BCCSP为加密服务提供者，为Fabric网络信息传输提供了密码学的保障。本章介绍了密码学相关知识及BCCSP概要，并对BCCSP源码进行了剖析。

第10章主要讲解了Fabric CA的架构设计。Fabric CA是Hyperledger Fabric行使证书机构的功能。本章介绍了Fabric CA用户指南，以及如何使用Fabric-CA-Server、Fabric-CA-Client等指令和HSM（硬件安全模块）。

第11章主要介绍了在Fabric中账本机制的设计与实现。本章的内容包括Ledger架构概述，Ledger中的Block-Storage、VersionedDB、HistoryDB等概念及其作用。

第12章主要通过具体的案例，加深读者对chaincode智能合约的理解，让读者了解Fabric中智能合约的编写方法。

第13章通过介绍一个完整的项目实例的实现来帮助读者更好地理解Fabric，让读者做到理论结合实际，进而提升实战经验。

第14章通过对官网案例进行部署，以体现源码和实际操作的呼应关系。

勘误与后续技术支持

由于作者的写作时间和水平有限，本书难免会存在一些纰漏和错误，欢迎广大读者批评指正。勘误请发送至作者邮箱：liangxiubo@hyperchain.cn。对于读者发现的问题，我们将在本书后续印次和版本中加以改正。

为了进一步降低区块链技术使用门槛，让更多的区块链开发者、爱好者以及正在尝试接入区块链技术的企业能够快捷地开发区块链应用，趣链科技于2017年9月14日正式上线了基于联盟链的“开发者平台”。基于该平台，用户可以更方便地创建、发布和使用多中心化的应用程序。通过平台提供的在线智能合约编辑器，用户可便捷、准确地编写智能合约程序；通过平台提供的区块链浏览器，用户可方便地获取链上区块信息、区块链节点状态、节点维护方信息等。欢迎广大区块链相关从业人员访问体验，开发者平台网址为：<https://dev.hyperchain.cn/>。

如需获得更多关于区块链技术的最新技术动态和趣链科技的技术支持，可关注微信公众号：hyperchain。

致谢

在此向所有给我们提供指导、支持和鼓励的朋友表示衷心的感谢。

感谢浙江大学计算机科学与技术学院和软件学院为我们提供的良好条件和各种便利，感谢陈纯院士、杨小虎研究员一直以来的关怀和支持。

感谢杭州趣链科技有限公司全体人员的大力支持，特别感谢李伟博士、李启雷博士、邱炜伟博士、尹可挺博士为本书成稿给予的鼎力支持，感谢曹靖、易洋溢、梁健、孟德佳、秦启睿、余清波、何昊等对书稿材料汇编所做的突出贡献，感谢余高成、周健、施泰龙等对书稿校阅所付出的时间和汗水。

感谢机械工业出版社华章公司的编辑们，他们不辞辛苦进行的仔细严谨的审阅和校对工作是本书顺利出版的有力保障。

蔡亮 梁秀波 宣章炯

Contents 目 录

序一	
序二	
前言	
第1章 准备工作	1
1.1 Go 语言环境配置	1
1.1.1 Go 语言简介	1
1.1.2 Go 安装	2
1.1.3 Go 标准包安装	4
1.1.4 第三方工具安装	6
1.1.5 Go 环境配置	7
1.1.6 代码目录结构规划	8
1.1.7 编译应用	9
1.1.8 获取远程包	10
1.1.9 程序的整体结构	11
1.2 安装 Docker	11
1.2.1 macOS	11
1.2.2 Ubuntu	12
1.2.3 Docker 的简易使用	13
1.3 Hyperledger 社区介绍	14
第2章 架构分析	18
2.1 Fabric 整体架构	18
2.1.1 概述	18
2.1.2 系统架构	19
2.1.3 交易背书的基本工作流程	24
2.1.4 背书策略	27
2.1.5 证实账本和节点账本检查	28
2.2 Fabric 交易流程	30
2.3 Fabric 整体项目结构介绍	33
2.3.1 Fabric 项目结构	33
2.3.2 Fabric 源码中相关缩写的含义	34
第3章 源码分析	37
3.1 Logging 日志模块浅析	37
3.1.1 go-logging 简介	37
3.1.2 flogging	38
3.1.3 init 函数、MustGetLogger 函数 与其他函数	38
3.2 Error 错误机制设计	39
3.2.1 总体概览	39
3.2.2 使用说明	40
3.2.3 显示错误消息	40
3.2.4 错误处理的一般准则	41
3.3 Config 配置模块的设计	41

3.3.1	viper 简介	41	5.2	kafka 排序服务机制讲解	79
3.3.2	安全文件配置	44	5.3	orderer 在 Fabric 中的交互流程	82
3.3.3	命令选项配置	44	5.3.1	建立连接	82
3.3.4	环境变量配置	44	5.3.2	Broadcast	83
3.4	grpc 服务	45	5.3.3	orderer	83
3.4.1	grpc 用法的 Demo	45	5.3.4	Deliver	86
3.4.2	Fabric 中的 grpc 服务接口和实例	46			
第 4 章	peer 的设计与实现	53	第 6 章	chaincode 的设计与实现	89
4.1	CommandLine 解析	53	6.1	chaincode 生命周期管理	89
4.1.1	peer 目录结构	53	6.1.1	打包	89
4.1.2	第三方包	54	6.1.2	安装 chaincode	91
4.1.3	peer 命令结构解析	55	6.1.3	实例化 chaincode	91
4.1.4	以 node 为例进行子命令结构解析	55	6.1.4	升级 chaincode	92
4.1.5	peer 命令结构	55	6.1.5	停止与启动	93
4.2	Admin 及 Endorser 服务的实现	56	6.1.6	CLI	93
4.2.1	Admin	56	6.2	chaincode 原理浅析	94
4.2.2	Endorser	58	6.2.1	什么是 chaincode	94
4.2.3	频道中的策略检查器	64	6.2.2	Chaincode Support 服务	95
4.3	Committer 的机制	66	6.2.3	FSM	95
4.3.1	committer.go 分析	66	6.2.4	Register	96
4.3.2	committer_impl.go 分析	67	6.2.5	Handler	97
4.3.3	validator.go 分析	70	6.2.6	processStream	97
4.3.4	vscv_validator.go 分析	71	6.2.7	HandleMessage	97
			6.2.8	serialSend 或 serialSendAsync	99
			6.2.9	系统 chaincode	99
第 5 章	order 的设计与实现	73	6.3	chaincode 数据结构分析	100
5.1	orderer 内部机制窥探	73	6.3.1	chaincode 元数据	100
5.1.1	kingpin	73	6.3.2	chaincode 的元工具	102
5.1.2	模块	74	6.4	SystemChaincode 讲解	103
5.1.3	配置	74	6.4.1	SystemChaincode	104
5.1.4	模块的初始化	75	6.4.2	预定义和注册	104

6.5	CSCC 分析	106	6.11.6	Execute	127
6.5.1	结构体	106	6.11.7	一路返回	128
6.5.2	函数	106	6.11.8	安装后的状态	129
6.6	ESCC 分析	108	6.12	ApplicationChaincode 的实例化	129
6.6.1	结构体	108	6.12.1	概述	129
6.6.2	Init 函数	108	6.12.2	起点	130
6.7	LSCC 分析	109	6.12.3	部署	130
6.7.1	结构体和接口	110	6.12.4	广播	139
6.7.2	函数操作	110	6.12.5	部署后的状态	139
6.7.3	安装、部署和升级	111	6.13	chaincode 操作步骤	140
6.7.4	chaincode stub 接口实现	112	6.13.1	选择一个代码存放位置	140
6.8	QSCC 分析	113	6.13.2	内务处理	140
6.8.1	结构体	113	6.13.3	初始化 chaincode	140
6.8.2	函数操作	114	6.13.4	调用 chaincode	142
6.8.3	路由规则	114	6.13.5	实现 chaincode 应用	143
6.9	VSCC 分析	115	6.13.6	整合全部代码	143
6.9.1	结构体	115	6.13.7	编译 chaincode	145
6.9.2	函数	115	6.13.8	在开发者模式下测试	145
6.10	SystemChaincode 的注册和实例化	116	6.13.9	安装 Hyperledger Fabric 样例	145
6.10.1	概述	116	6.13.10	下载 Docker 镜像	146
6.10.2	安装	117	6.13.11	1 号终端	146
6.10.3	部署	117	6.13.12	2 号终端	146
6.10.4	Launch	118	6.13.13	3 号终端	147
6.10.5	Execute	123	6.13.14	测试新的 chaincode	147
6.10.6	部署后状态	124	第 7 章	MSP 成员服务提供者	148
6.11	ApplicationChaincode 的部署	124	7.1	MSP 的设计思路	148
6.11.1	概述	125	7.1.1	MSP 配置	149
6.11.2	生成签名申请包	125	7.1.2	如何生成 MSP 证书和它们的 签名匙	150
6.11.3	处理安装申请	125	7.1.3	MSP setup on the peer & orderer side	150
6.11.4	执行申请	126			
6.11.5	Launch	127			

7.1.4	Channel MSP setup	150
7.1.5	最佳实践	151
7.2	MSP 实现剖析	153
7.2.1	目录结构	153
7.2.2	MSP 配置	154
第 8 章	Gossip 节点间的流言蜚语	162
8.1	Gossip 协议原理解析	162
8.1.1	Gossip 协议 (Gossip protocol)	162
8.1.2	Gossip 消息传输 (Gossip messaging)	163
8.2	Gossip 之服务组件	163
8.2.1	protos/gossip 分析	163
8.2.2	Gossip 服务组件	169
8.2.3	gossip 消息发送方式详解	176
8.3	Gossip 之服务初始化	178
8.3.1	gossipSvc 组件	179
8.3.2	chains 组件	185
8.3.3	leaderElection 组件	187
8.3.4	gossip 服务的停止	193
8.4	Gossip 之消息广播	194
8.4.1	gossip 服务消息的散播过程	194
8.4.2	消息从何而来	194
8.4.3	消息如何散播	196
8.4.4	消息去往何方	200
8.5	channel 通道的设计与实现	201
8.5.1	概述	201
8.5.2	配置文件	202
8.5.3	命令	203
8.6	事件机制	208
8.6.1	Fabric 中 Event 相关实现	208
8.6.2	events/producer	209

8.6.3	Go SDK 中 Event 相关实现	210
-------	---------------------	-----

第 9 章 BCCSP 加密服务提供者的设计与实现

9.1	密码学相关知识介绍	212
9.1.1	安全基础	212
9.1.2	加密基础	213
9.1.3	哈希函数	214
9.1.4	共享密钥加密	214
9.1.5	公钥加密	215
9.1.6	混合加密	216
9.1.7	消息验证码	216
9.1.8	数字签名	218
9.1.9	数字证书	219
9.2	BCCSP 概要	220
9.2.1	BCCSP 简介	220
9.2.2	陷阱函数	222
9.2.3	为什么要使用 ECDSA	223
9.2.4	生成签名	223
9.2.5	验证签名	224
9.3	BCCSP 源码剖析	224
9.3.1	BCCSP 服务结构	224
9.3.2	BCCSP 中的接口和选项	225
9.3.3	SW 实现方式	227
9.3.4	pkes11 实现方式	230

第 10 章 Fabric CA 架构设计与讲解

10.1	Fabric CA 用户指南	233
10.2	Fabric-CA-Server	240
10.2.1	初始化服务端	241
10.2.2	算法和密钥长度	242

10.2.3	启动服务端	243	11.2	Ledger 之 Block-Storage	275
10.2.4	配置数据库	243	11.2.1	peer 节点中的 leveldb	276
10.2.5	PostgreSQL	243	11.2.2	peer 节点中的账本	276
10.2.6	PostgreSQL SSL 配置	244	11.2.3	创建	276
10.2.7	MySQL	245	11.2.4	使用	277
10.2.8	MySQL SSL 配置	245	11.2.5	idStore	278
10.2.9	配置 LDAP	246	11.2.6	存储账本 ID	278
10.2.10	构建一个集群	249	11.2.7	ConstructionFlag	278
10.2.11	构建多个 CA	249	11.2.8	账本恢复	279
10.2.12	登录一个中间 CA	250	11.2.9	BlockStore	280
10.2.13	升级服务端	251	11.3	Ledger 之 VersionedDB	286
10.2.14	升级一个集群	251	11.3.1	peer 节点使用 VersionedDB	286
10.3	fabric-ca-client	253	11.3.2	交易模拟器 / 交易查询器	288
10.3.1	登录启动用户	253	11.3.3	重启恢复	298
10.3.2	注册一个新身份	253	11.4	Ledger 之 HistoryDB	300
10.3.3	登录一个节点	256	11.4.1	历史查询器	300
10.3.4	从另一个 Fabric CA 服务器 获得 CA 证书链	257	11.4.2	使用	301
10.3.5	重新登录一个身份	257	第 12 章	chaincode 智能合约案例 分析	303
10.3.6	撤销一个证书或身份	257	12.1	encc_example	303
10.3.7	生成一个 CRL	259	12.1.1	chaincode 代码分析	303
10.3.8	启用 TLS	259	12.1.2	使用 EncCC	307
10.3.9	基于属性的访问控制	260	12.2	eventsender	308
10.3.10	动态更新服务器配置	261	12.3	example01	310
10.3.11	联系特定的 CA 实例	265	12.4	example02	311
10.4	HSM	265	12.5	example03	314
第 11 章	账本机制的设计与实现	267	12.6	example04	315
11.1	Ledger 架构概述	267	12.7	example05	317
11.1.1	总览	267	12.8	invokereturnsvalue	319
11.1.2	ledger 部分摘要	268	12.9	map	320
			12.10	marbles02	324

12.11	passthru	332	13.4.5	更新账本	358
12.12	sleeper	332	13.5	Balance transfer	360
第 13 章 Fabric-samples 项目分析与实践					
13.1 Fabric-samples 项目结构					
13.2 First-network					
13.2.1	安装预置环境	335	13.5.1	预置环境	360
13.2.2	想要现在运行吗?	335	13.5.2	工件	360
13.2.3	生成网络神器	336	13.5.3	运行示例程序	360
13.2.4	启动网络	337	13.5.4	示例——REST APIs 请求	361
13.2.5	关闭网络	339	13.6	Hyperledger Fabric CA 示例	365
13.2.6	加密生成器	339	13.6.1	运行这个示例	366
13.2.7	配置交易生成器	340	13.6.2	了解这个例子	366
13.2.8	运行工具	341	13.7	高性能网络	367
13.2.9	启动网络	342	13.7.1	用例	368
13.2.10	了解 Docker Compose 技术	347	13.7.2	如何使用	368
13.2.11	使用 CouchDB	348	第 14 章 部署教程		
13.2.12	关于数据持久化的提示	350	14.1	下载部署环境	371
13.2.13	故障排除	350	14.2	编译 peer、orderer、configtxgen 等程序	373
13.3	basic-network	351	14.3	部署	374
13.4	Fabcar	353	14.4	Crypto Generator	374
13.4.1	编写第一个应用	353	14.4.1	crypto-config.yaml	375
13.4.2	下载测试网络 (Getting a Test Network)	354	14.4.2	crypto-config 文件夹	375
13.4.3	应用程序如何与网络进行交互	355	14.5	Configuration Transaction Generator	376
13.4.4	查询账本	355	14.6	networkUp- 启动 Fabric 网络	378
			14.7	运行容器 + 区域链操作	378
			附录 专业术语		
			381		



第 1 章 Chapter 1

准备工作

本章将介绍如何从零开始，通过准备工作完成多种平台上 Hyperledger Fabric 应用环境的配置。本章首先会讲解不同系统中 Go 语言的安装与环境配置，并简单介绍了 Go 语言；之后会讲解不同系统中 Docker 的安装和简易使用；最后将介绍 Hyperledger 社区与社区中一些常用软件，方便读者在完成环境配置之余，更好地了解 Hyperledger Fabric 的相关信息。

1.1 Go 语言环境配置

本节详细讲解了不同系统下 Go 语言与其第三方工具的多种配置方式，使读者在不同系统下顺利完成配置。

1.1.1 Go 语言简介

Go 是从 2007 年年末由 Robert Griesemer、Rob Pike、Ken Thompson 主持开发，后来在开发团队还加入了 Ian Lance Taylor、Russ Cox 等人，并最终于 2009 年 11 月开源，在 2012 年早些时候发布了 Go 1 稳定版本。现在 Go 的开发已经是完全开放的了，并且其拥有一个十分活跃的社区。

作为一种新的、并发的、带垃圾回收和快速编译的语言，Go 具有以下特点：

它可以在一台计算机上用几秒钟的时间编译一个大型的 Go 程序。Go 为软件构造提供了一种模型，它使得依赖分析更加容易，且避免了大部分 C 风格 include 文件与库的开头。

Go 是静态类型的语言，它的类型系统没有层级。因此用户不需要在定义类型之间的关系上花费时间，这比起典型的面向对象语言会更轻量级。

作为垃圾回收型的语言，Go 为并发执行与通信提供了基本的支持。同时按照其设计，Go 准备为多核机器上系统软件的构造提供一种新的方法。

Go 是一种编译型语言，它结合了解释型语言的游刃有余，动态类型语言的开发效率，以及静态类型的安全性。开发者也打算把 Go 打造成现代的、支持网络与多核计算的语言。Go 满足了上面这些目标，同时解决一些语言上的问题，比如说并发与垃圾回收机制，严格的依赖规范等。

Hyperledger Fabric 采用 Go 语言开发，因此，在本章中，我们将分别介绍 Go 语言相关的各种知识背景，便于读者后续理解。如图 1-1 所示为 Go 语言 logo。

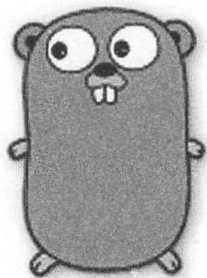


图 1-1 Go 语言 logo

1.1.2 Go 安装

读者可根据自身喜好，以及使用系统等实际情况选择 Go 的安装方式。

1. Go 的三种安装方式

Go 有多种安装方式，读者可以选择自己喜欢的方式。这里介绍三种最常见的安装方式：

- ❑ Go 源码安装：这是一种标准的软件安装方式。对于经常使用 Unix 类系统的用户，尤其对于开发者来说，从源码安装可以修改个性化软件配置。
- ❑ Go 标准包安装：Go 提供了方便的安装包，支持 Windows、Linux、Mac 等系统。这种方式适合快速安装，可根据自己的系统位数下载好相应的安装包，默认设置后就可以轻松安装了，推荐使用这种方式。
- ❑ 第三方工具安装：目前有很多方便的第三方软件包工具，例如 Ubuntu 的 apt-get 和 wget、Mac 的 homebrew 等。这种安装方式适合那些熟悉相应系统的用户。最后，如果你想在同一个系统中安装多个版本的 Go，你可以参考第三方工具 GVM，这是目前在这方面做得最好的工具。

2. Go 源码安装

Go 1.5 彻底移除了 C 代码，Runtime、Compiler、Linker 均由 Go 编写，实现自举。只需要安装了上一个版本的 Go，即可从源码安装。

在 Go 1.5 前的源代码中，有些部分是用 Plan 9 C 和 AT&T 汇编写的，因此假如你想要从源码安装，就必须安装 C 的编译工具。

在 Mac 系统中，只要你安装了 Xcode，就已经包含了相应的编译工具。

在类 Unix 系统中，需要安装 gcc 等工具。例如 Ubuntu 系统可通过在终端中执行 `sudo apt-get install gcc libc6-dev` 来安装编译工具。

在 Windows 系统中，你需要安装 MinGW，然后通过 MinGW 安装 gcc，并设置相应环境变量。

你可以直接去官网下载源码，找相应的 `goVERSION.src.tar.gz` 的文件下载，下载之后解压到 \$HOME 目录下，执行如下代码：

```
cd go/src
./all.bash
```

运行 all.bash 后出现 “ALL TESTS PASSED” 字样才算安装成功。

上面是 Unix 风格的命令，Windows 系统下的安装方式类似，只不过是运行 all.bat，调用的编译器是 MinGW 的 gcc。

除此之外，如果是 Mac 或者 Unix 用户还需要设置若干个环境变量来使系统识别 go 命令位置，如果想重启之后也能生效 go 命令的话，则把下面的命令写到 .bashrc 或者 .zshrc 里面：

```
export GOPATH=$HOME/gopath
export PATH=$PATH:$HOME/go/bin:$GOPATH/bin
```

如果你将以上两行命令写入了 .bashrc 或者 .zshrc 中，执行 bash.bashrc 或者 bash.zshrc 可以使得设置立刻生效。

如果是 Window 系统，就需要设置环境变量，在 path 里面增加相应的 go 所在的目录，设置 gopath 变量。

当你设置完毕之后在命令行里面输入 go，看到图 1-2 所示结果即说明已经安装成功。

```
→ ~ go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build      compile packages and dependencies
    clean      remove object files
    env        print Go environment information
    fix        run go tool fix on packages
    fmt        run gofmt on package sources
    generate    generate Go files by processing source
    get         download and install packages and dependencies
    install     compile and install packages and dependencies
    list        list packages
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c          calling between Go and C
    filetype   file types
    gopath     GOPATH environment variable
    importpath import path syntax
    packages   description of package lists
    testflag   description of testing flags
    testfunc   description of testing functions

Use "go help [topic]" for more information about that topic.
```

图 1-2 Go 语言成功配置

如上所示,如果出现 Go 的 Usage 信息,那么说明 Go 已经安装成功了;如果出现该命令不存在,那么可以检查一下自己的 PATH 环境变量中是否包含了 Go 的安装目录。

从 Go 1.8 开始, GOPATH 环境变量现在有一个默认值。如果它没有被设置,它在 Unix 上默认为 \$HOME/go, 在 Windows 上默认为 %USERPROFILE%/go。

1.1.3 Go 标准包安装

Go 提供了每个平台打好包的一键安装,这些包默认会安装到如下目录: /usr/local/go (Windows 系统: c:\Go) 中,当然你可以改变它们的安装位置,但是改变之后你必须在你的环境变量中设置如下信息:

```
export GOROOT=$HOME/go
export GOPATH=$HOME/gopath
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

上面这些命令对于 Mac 和 Unix 用户来说最好是写入 .bashrc 或者 .zshrc 文件,对于 Windows 用户来说需要写入环境变量。

1. 如何判断自己的操作系统是 32 位还是 64 位?

接下来的 Go 安装需要判断操作系统的位数,所以需应确定自己的系统类型。

Windows 系统用户请按 Win+R 运行 cmd, 输入 systeminfo 后回车,稍等片刻,会出现一些系统信息。在“系统类型”一行中,若显示“x64-based PC”,即为 64 位系统;若显示“X86-based PC”,则为 32 位系统。

Mac 系统用户建议直接使用 64 位的,因为 Go 支持的 Mac OS X 版本已经不支持纯 32 位处理器了。

Linux 系统用户可通过在 Terminal 中执行命令 arch(即 uname -m)来查看系统信息: 64 位系统显示 x86_64, 32 位系统显示 i386。

2. Mac 安装

访问下载地址, 32 位系统下载 go1.4.2.darwin-386-osx10.8.pkg (更新的版本已无 32 位下载), 64 位系统下载 go1.8.3.darwin-amd64.pkg, 双击下载文件, 默认安装点击下一步, 这个时候 Go 已经安装到你的系统中, 默认已经在 PATH 中增加了相应的 ~/go/bin, 这个时候打开终端, 输入 Go。

如果出现 Go 的 Usage 信息,那么说明 Go 已经安装成功了;如果出现该命令不存在,那么可以检查一下自己的 PATH 环境变中是否包含了 Go 的安装目录。

3. Linux 安装

访问下载地址, 32 位系统下载 go1.8.3.linux-386.tar.gz, 64 位系统下载 go1.8.3.linux-amd64.tar.gz, 假定你想要安装 Go 的目录为 \$GO_INSTALL_DIR, 后面替换为相应的目录路径即可。

解压缩 tar.gz 包到安装目录下: `tar zxvf go1.8.3.linux-amd64.tar.gz -C $GO_INSTALL_DIR`。设置 PATH, `export PATH=$PATH:$GO_INSTALL_DIR/go/bin`。然后执行 go, 如图 1-3 所示。

```

→ ~ go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build      compile packages and dependencies
    clean      remove object files
    env        print Go environment information
    fix        run go tool fix on packages
    fmt        run gofmt on package sources
    generate    generate Go files by processing source
    get        download and install packages and dependencies
    install    compile and install packages and dependencies
    list       list packages
    run        compile and run Go program
    test       test packages
    tool       run specified go tool
    version    print Go version
    vet        run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

    c          calling between Go and C
    filetype   file types
    gopath     GOPATH environment variable
    importpath import path syntax
    packages   description of package lists
    testflag   description of testing flags
    testfunc   description of testing functions

Use "go help [topic]" for more information about that topic.

```

图 1-3 Go 成功安装

如果出现 Go 的 Usage 信息, 那么说明 Go 已经安装成功了; 如果出现该命令不存在, 那么可以检查一下自己的 PATH 环境变量中是否包含了 Go 的安装目录。

4. Windows 安装

访问 Golang 下载页, 32 位请选择名称中包含 windows-386 的 msi 安装包, 64 位请选择名称中包含 windows-amd64 的 msi 安装包。下载好后运行, 不要修改默认安装目录 C:\Go\, 若安装到其他位置会导致不能执行自己编写的 Go 代码。安装完成后默认会在环境变量 Path 后添加 Go 安装目录下的 bin 目录 C:\Go\bin\, 并添加环境变量 GOROOT, 值为 Go 安装根目录 C:\Go\。

验证是否安装成功

在运行中输入 `cmd` 打开命令行工具，在提示符下输入 `go`，检查是否能看到 `Usage` 信息。输入 `cd %GOROOT%`，看能否进入 `Go` 安装目录。若都成功，说明安装成功。

不能的话请检查上述环境变量 `Path` 和 `GOROOT` 的值。若不存在请卸载后重新安装，存在请重启计算机后重试以上步骤。

1.1.4 第三方工具安装

使用第三方工具可以大大降低软件管理的难度，读者可根据习惯与喜好选择工具。

1. gvm

`gvm` 是第三方开发的 `Go` 多版本管理工具，类似 `ruby` 里面的 `rvm` 工具。使用起来相当方便，安装 `gvm` 使用如下命令：

```
bash < <(curl -s -S -L https://raw.githubusercontent.com/moovweb/gvm/master/
  binscripts/gvm-installer)
```

安装完成后，就可以安装 `Go` 了：

```
gvm install go1.8.3
gvm use go1.8.3
```

也可以使用下面的命令，省去每次调用 `gvm use` 的麻烦：`gvm use go1.8.3 --default`。

执行完上面的命令之后，`GOPATH`、`GOROOT` 等环境变量会自动设置好，这样就可以直接使用了。

2. apt-get

`Ubuntu` 是目前使用最多的 `Linux` 桌面系统，使用 `apt-get` 命令来管理软件包，可以通过下面的命令来安装 `Go`，为了以后方便，应该把 `git` `mercurial` 也安装上：

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:gophers/go
sudo apt-get update
sudo apt-get install golang-stable git-core mercurial
```

3. wget

安装 `wget` 使用如下命令：

```
wget https://storage.googleapis.com/golang/go1.8.3.linux-amd64.tar.gz
sudo tar -xzf go1.8.3.linux-amd64.tar.gz -C /usr/local
```

配置环境变量：

```
export GOROOT=/usr/local/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
export GOPATH=$HOME/gopath （可选设置）
```


或者使用：

```
sudo vim /etc/profile
```

并添加下面的内容：

```
export GOROOT=/usr/local/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
export GOPATH=$HOME/gopath （可选设置）
```

重新加载 profile 文件

```
source /etc/profile
```

4. homebrew

homebrew 是 Mac 系统下目前使用最多的管理软件的工具，目前已支持 Go，可以通过命令直接安装 Go，为了以后方便，应该把 git mercurial 也安装上：

(1) 安装 homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

(2) 安装 go

```
brew update && brew upgrade
brew install go
brew install git
brew install mercurial //可选安装
```

1.1.5 Go 环境配置

Go 环境配置有如下步骤：

创建 GOPATH 目录，在目录下创建 src、pkg、bin 三个子目录。src 目录为放置源代码以及第三方包的目录，pkg 为编译生成的一些静态链接库所存放的目录，bin 则为 go install 编译生成的二进制文件所存放的位置。然后设置相应的环境变量。

Mac 或者 Linux 系统用户可在 .bash_profile 或者 .bashrc 中进行设置。

```
cd ~
vim .bash_profile
```

将 bash 文件修改为：

```
export GOROOT=YOURPATH/go
export GOPATH=YOURPATH/YOURGOPATH
export PATH=${PATH}:$GOROOT/bin:$GOPATH/bin
```

然后在命令行中测试：

```
go version
#go version go1.9 darwin/amd64
```

如果出现以上信息，则说明 Go 语言环境搭建成功。

1. GOPATH 与工作空间

在 1.1.2~1.1.4 节中安装 Go 的时候看到需要设置 GOPATH 变量，Go 从 1.1 版本到 1.7 版本必须设置这个变量，环境变量 GOPATH 中包含的路径不能和 Go 的安装目录一样，这个目录用来存放 Go 源码、Go 的可运行文件，以及相应的编译之后的包文件。所以这个目录下面有三个子目录：src、bin、pkg。

从 go 1.8 开始，GOPATH 环境变量现在有一个默认值，如果它没有被设置，则它在 Unix 上默认为 \$HOME/go，在 Windows 上默认为 %USERPROFILE%/go。

2. GOPATH 设置

Go 命令依赖一个重要的环境变量：\$GOPATH。

Windows 系统中环境变量的形式为 %GOPATH%，本书主要使用 Unix 形式，Windows 用户请自行替换。

在类 Unix 环境下设置如下：

```
export GOPATH=/home/apple/mygo
```

为了方便，应该新建以上文件夹，并且将上面一行加入到 .bashrc、.zshrc 或者自己的 sh 的配置文件中。

Windows 设置如下，新建一个叫作 GOPATH 的环境变量：

```
GOPATH=c:\mygo
```

GOPATH 允许多个目录，当有多个目录时，请注意分隔符，多个目录的时候 Windows 是分号，Linux 系统是冒号。当有多个 GOPATH 时，默认会将 go get 的内容放在第一个目录下。

以上 \$GOPATH 目录约定有三个子目录：

- ❑ src 存放源代码（比如：.go .c .h .s 等）
- ❑ pkg 编译后生成的文件（比如：.a）
- ❑ bin 编译后生成的可执行文件（为了方便，可以把此目录加入到 \$PATH 变量中，如果有多个 gopath，那么使用 \${GOPATH//:/bin:}/bin 添加所有的 bin 目录）

1.1.6 代码目录结构规划

GOPATH 下的 src 目录就是接下来开发程序的主要目录，所有的源码都放在这个目录下，一般的做法是一个目录匹配一个项目，例如：\$GOPATH/src/mymath 表示 mymath 这个应用包或者可执行应用，这个根据 package 是 main 还是其他来决定，main 的话就是可执行应用，其他的话就是应用包，package 会在后续详细介绍。

所以当新建应用或者一个代码包时，则在 src 目录下新建一个文件夹，文件夹名称一般是代码包名称，当然也允许多级目录，例如在 src 下面新建了目录 \$GOPATH/src/github。

com/astaxie/beedb, 那么这个包路径就是“github.com/astaxie/beedb”, 包名称是最后一个目录 beedb。

下面就以 mymath 为例来讲述如何编写应用包, 执行如下代码:

```
cd $GOPATH/src
mkdir mymath
```

新建文件 sqrt.go, 内容如下:

```
// $GOPATH/src/mymath/sqrt.go源码如下:
package mymath
func Sqrt(x float64) float64 {
    z := 0.0
    for i := 0; i < 1000; i++ {
        z -= (z*z - x) / (2 * x)
    }
    return z
}
```

这样应用包目录和代码已经新建完毕, 需要注意的是, 一般建议 package 的名称和目录名保持一致。

1.1.7 编译应用

上面已经建立了自己的应用包, 那么如何进行编译安装呢? 有两种方式可以选择:

- 1) 进入对应的应用包目录, 然后执行 go install
- 2) 在任意的目录执行代码 go install mymath

安装完之后, 可以进入如下目录:

```
cd $GOPATH/pkg/${GOOS}_${GOARCH}
//可以看到如下文件
mymath.a
```

这个 .a 文件是应用包, 新建一个应用程序来调用这个应用包 mathapp:

```
cd $GOPATH/src
mkdir mathapp
cd mathapp
vim main.go
$GOPATH/src/mathapp/main.go源码:
package main

import (
    "mymath"
    "fmt"
)

func main() {
    fmt.Printf("Hello, world. Sqrt(2) = %v\n", mymath.Sqrt(2))
}
```


可以看到这个 package 是 main, import 里面调用的包是 mymath, 这就是相对于 \$GOPATH/src 的路径, 如果是多级目录, 就在 import 里面引入多级目录。如果你有多个 GOPATH, 也是一样, Go 会自动在多个 \$GOPATH/src 中寻找。

如何编译程序呢? 进入该应用目录, 然后执行 go build, 在该目录下面会生成一个 mathapp 的可执行文件:

```
./mathapp
```

输出如下内容:

```
Hello, world. Sqrt(2) = 1.414213562373095
```

安装该应用需要进入该目录执行 go install, 然后在 \$GOPATH/bin/ 下增加一个可执行文件 mathapp, 前面已经把 \$GOPATH/bin 加到 PATH 里面了, 这样在命令行输入如下命令就可以执行:

```
mathapp
```

输出如下内容:

```
Hello, world. Sqrt(2) = 1.414213562373095
```

上面展示了如何编译和安装一个可运行的应用, 以及如何设计目录结构。

1.1.8 获取远程包

Go 语言有一个获取远程包的工具——go get, 目前 go get 支持多数开源社区 (例如: Github、Googlecode、bitbucket、Launchpad)

```
go get github.com/astaxie/beedb
```

go get -u 参数可以自动更新包, 而且 go get 会自动获取该包依赖的其他第三方包。

通过上面的命令可以获取相应的源码, 对应的开源平台采用不同的源码控制工具, 例如 Github 采用 git, Googlecode 采用 hg, 所以要想获取这些源码, 必须先安装相应的源码控制工具。

上面获取的代码在本地源码相应的代码结构如下:

```
$GOPATH
src
|--github.com
|   |--astaxie
|       |--beedb
pkg
|--相应平台
|   |--github.com
|       |--astaxie
|           |--beedb.a
```



`go get` 可以理解为以下 2 步：第一步是通过源码工具 `clone` 从互联网下载或更新指定的代码包及其依赖包到 `src` 下面；第 2 步执行 `go install` 对代码包及其依赖包进行编译和安装。

在代码中使用远程包的方法和使用本地包一样，只要在开头 `import` 相应的路径就可以了：

```
import "github.com/astaxie/beedb"
```

1.1.9 程序的整体结构

通过上面建立的本地的 `mygo` 的目录结构如下所示：

```
bin/  
  mathapp  
pkg/  
  平台名/ 如: darwin_amd64、linux_amd64  
    mymath.a  
    github.com/  
      astaxie/  
        beedb.a  
src/  
  mathapp  
    main.go  
  mymath/  
    sqrt.go  
  github.com/  
    astaxie/  
      beedb/  
        beedb.go  
        util.go
```

从上面的结构，可以很清晰地看到，`bin` 目录下面存放的是编译之后可执行的文件，`pkg` 下面存放的是应用包，`src` 下面存放的是应用源代码。

1.2 安装 Docker

Docker 是一个开源的应用容器引擎，基于 Go 语言并遵从 Apache2.0 协议开源，让开发者可以将他们的应用以及依赖包打包到一个可移植的容器中，然后在任意主流的 Linux 机器上运行，也可以实现虚拟化，容器完全使用沙箱机制，相互之间不会有任何接口。

1.2.1 macOS

macOS 环境下，通过官网下载相应的 `.dmg` 文件即可，链接为：<https://store.docker.com/editions/community/docker-ce-desktop-mac>。

双击安装，点击相应的图标启动，在命令行中测试：



12 ❖ Hyperledger Fabric 源代码分析与深入解读

```
docker version
#Client:
# Version:      17.09.0-ce
# API version:   1.32
# Go version:    go1.8.3
# Git commit:    afd6b6d4
# Built:         Tue Sep 26 22:40:09 2017
# OS/Arch:       darwin/amd64

#Server:
# Version:      17.09.0-ce
# API version:   1.32 (minimum version 1.12)
# Go version:    go1.8.3
# Git commit:    afd6b6d4
# Built:         Tue Sep 26 22:45:38 2017
# OS/Arch:       linux/amd64
# Experimental:  true
```

由于国内下载镜像速度较慢，所以建议配置相应的加速器。

在菜单中点击“Preferences...”，然后查看 Daemon 标签，在 Registry mirrors 中可以点击“+”来添加加速器地址，一般可以选择阿里云的镜像加速器地址或者 daocloud 的镜像加速器。

1.2.2 Ubuntu

Docker 要求 Ubuntu 系统的内核版本高于 3.10，安装前先验证 Ubuntu 版本是否支持 Docker。在 Ubuntu 系统下，有多种 Docker 安装方式可供选择，如果当前安装方式速度较慢或者失败，不妨换一种试试。

如果安装了旧的 Docker，需要先卸载旧版本。

```
$ sudo apt-get remove docker docker-engine docker.io
```

安装新的 Docker 的方法有三种，分别是 repository 安装，从安装包安装和通过脚本快速安装。

(1) 从 repository 安装

具体步骤如下所示：

1) 更新源

```
sudo apt-get update
```

2) 安装包

```
sudo apt-get install \
  apt-transport-https \
  ca-certificates \
  curl \
  software-properties-common
```




3) 添加 Docker 官方的 GPG key

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

4) 向 source.list 中添加 Docker 软件源

```
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

5) 安装 Docker CE

更新 apt 软件包缓存，并安装 docker-ce:

```
sudo apt-get update
sudo apt-get install docker-ce
```

(2) 从安装包安装

从 <https://download.docker.com/linux/ubuntu/dists/>，选择你对应的 Ubuntu 版本。

安装 Docker CE，将路径改为你下载 Docker 安装包的路径。安装代码如下：

```
sudo dpkg -i /path/to/package.deb
```

(3) 通过脚本快速安装

脚本快速安装代码如下：

```
curl -fsSL get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

1.2.3 Docker 的简易使用

Docker 的简易使用方法如下：

(1) 使用非 root 用户管理 Docker

默认情况下，Docker 通过 Unix socket 与 Docker 引擎通讯，只有 root 用户和 Docker 组的用户才可以访问，一般建议将需要使用 Docker 的用户加入 Docker 用户组。

建立 Docker 组：

```
sudo groupadd docker
```

将当前用户加入 Docker 组：

```
sudo usermod -aG DOCKER $USER
```

退出当前用户，重新登录后生效。

(2) 镜像加速器

在 `/etc/docker/daemon.json` 中添加你所需要的加速器地址：

```
{
    "registry-mirrors": ["https://registry.docker-cn.com"]
}
```



(3) 卸载 Docker CE

卸载 Docker CE 包:

```
sudo apt-get purge docker-ce
```

镜像、容器、数据卷以及自定义的配置文件都存放在你的主机上, 需要手动移除:

```
sudo rm -rf /var/lib/docker
```

(4) 重启 Docker 服务

可以使用:

```
sudo service docker restart
```

或者

```
sudo systemctl restart docker
```

1.3 Hyperledger 社区介绍

Hyperledger 是一个全球跨行业领导者的商业区块链技术合作项目, 由 Linux 基金会主管, 领导者囊括了金融、银行、物联网、供应链、制造和技术领域的佼佼者。

Hyperledger 社区介绍

Hyperledger 社区主要由两大部分组成:

项目: 关注区块链架构和模块的实现。

工作组: 关注其他方面, 主要掌握 Hyperledger 社区项目的推广及日常工作。

如图 1-4 所示为 Hyperledger 社区。

下面我们将介绍 Hyperledger 的相关知识点。

(1) 工作组

- ❑ Technical Steering Committee (技术委员会)
- ❑ Architecture Working Group (架构委员会)
- ❑ Identity Working Group (身份工作组)
- ❑ Performance and Scale Working Group (性能和可伸缩性工作组)
- ❑ Whitepaper Working Group (白皮书工作组)
- ❑ Technical Working Group China (TWG-China, 大中华区技术工作组)
- ❑ Training and Education Working Group (培训与教育工作组)

(2) 相关文档

- ❑ Fabric 的官方文档: <http://hyperledger-fabric.readthedocs.io/en/latest/>
- ❑ 相应的设计文档: <https://wiki.hyperledger.org/projects/fabric/design-docs>



- ❑ 会议记录: <https://wiki.hyperledger.org/projects/fabric/meeting-notes/start>
- ❑ 相关视频: https://www.youtube.com/channel/UCCFdgcCWH_1vCndMPVqQlwZw

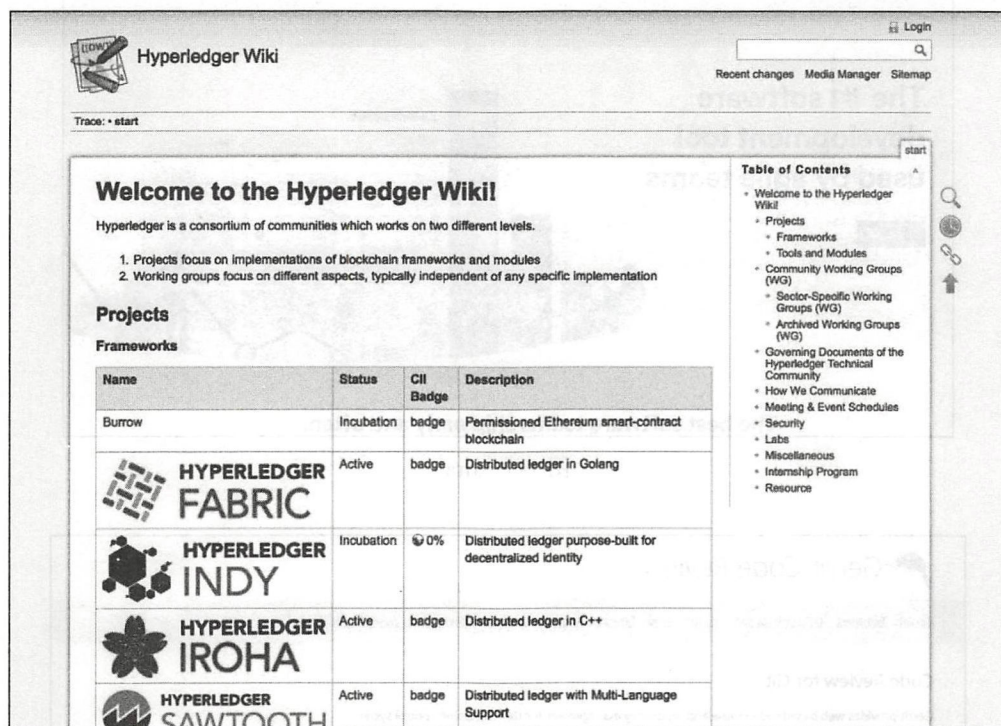


图 1-4 Hyperledger 社区

(3) Linux Foundation ID

Linux Foundation ID 是登录 Linux 基金会相关平台的唯一身份标识, 超级账本项目是 Linux 基金会下的项目, 可以在 <https://identity.linuxfoundation.org> 上注册。这个账户可以用于登陆 Jira、Gerrit、RocketChat 等平台。

(4) Jira

Jira 是 Atlassian 公司出品的项目与事务跟踪工具, 被广泛应用于缺陷跟踪、客户服务、需求收集、流程审批、任务跟踪、项目跟踪和敏捷管理等工作领域, 超级账本通过 Jira 来实现具体需求的追踪, 以此来更好地管理维护整个 Hyperledger 项目, 如图 1-5 所示。

(5) Gerrit

Gerrit 是一种免费、开源的代码审查软件, 使用网页界面, 如图 1-6 所示。利用网页浏览器, 同一个团队的软件程序员可以相互审阅彼此修改后的程序代码, 进而决定提交、退回或者继续修改。它使用 Git 作为底层版本控制系统。超级账本通过 Gerrit 来进行代码的提交及 review 工作。

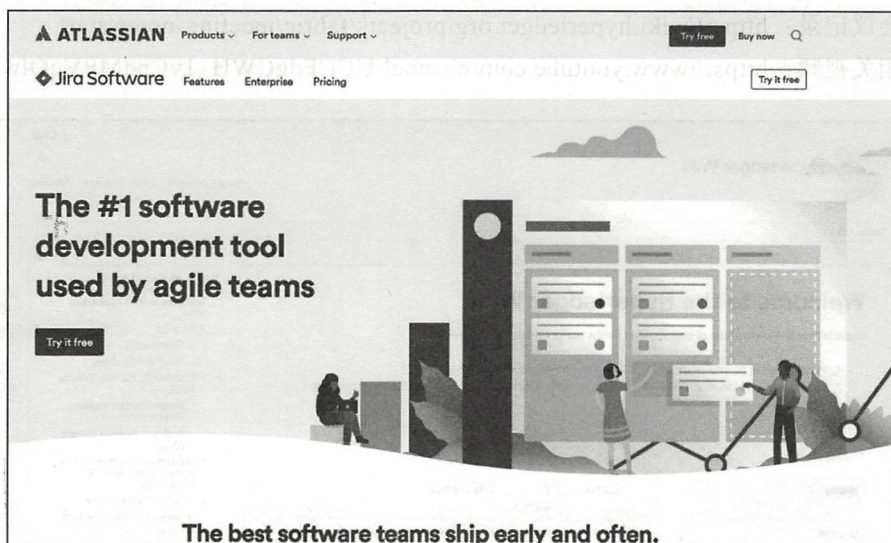


图 1-5 Jira

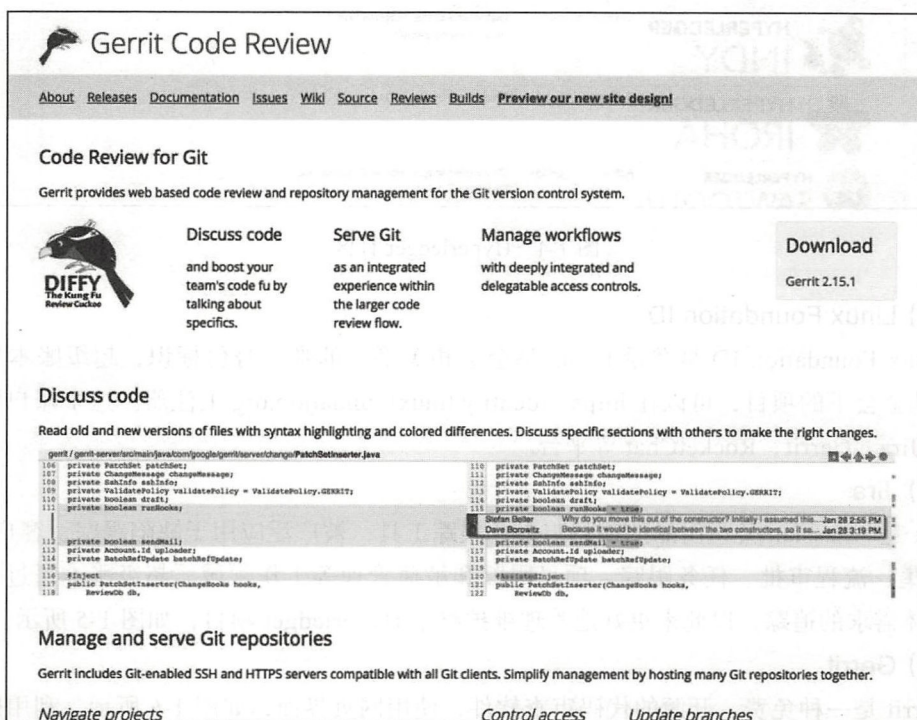


图 1-6 Gerrit

(6) 交流

超级账本采用 RocketChat 来作为社区成员间的沟通交流的工具，使世界各地的开发者





都能够公开参与相关问题的研讨，如图 1-7 所示。

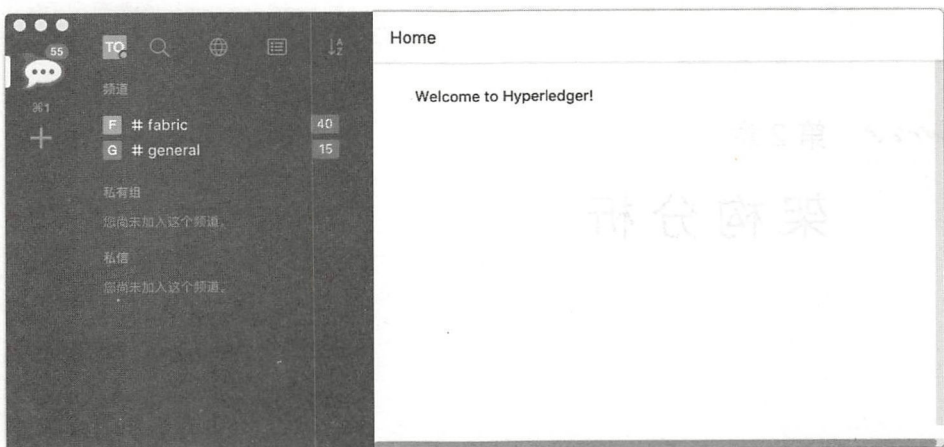
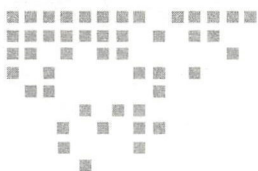


图 1-7 RocketChat





Chapter 2 第2章

架构分析

本章主要讲解 Fabric 的架构并进行分析。

2.1 Fabric 整体架构

Hyperledger Fabric 是一个区块链框架实现，也是 Linux 基金会托管的 Hyperledger 项目之一。作为使用模块化体系结构开发应用程序或解决方案的基础，Hyperledger Fabric 允许组件（如共识和会员服务）即插即用。

2.1.1 概述

FabricV1 版本相对于 0.6 版本实现了以下改进。

- ❑ Chaincode 信任的灵活性：Fabric1.0 架构将链码的信任假设和排序的信任假设进行了拆分。换句话说，排序服务可以由一组排序节点提供，具有一定容错能力，而且可以灵活指定链码的背书策略。
- ❑ 可扩展性：作为特定链码的背书节点和排序节点是垂直交叉关系，这使得这些功能的系统性能比在同一节点上完成得更好。特别是当不同的链码指定不同的背书者时更是如此，为此在背书者之间引入了链码分区技术，允许链码并行执行（和背书）。此外，链码的执行，可能比较耗费计算资源，所以把它从排序服务的关键路径移除。
- ❑ 机密性：该架构利于有保密性要求的链码部署，能满足交易内容和状态更新的保密性要求。
- ❑ 共识模块化：该架构是模块化的，允许可插拔的模块化共识（即排序服务）实现。

如图 2-1 所示的是 Fabric-v0.6 的架构。



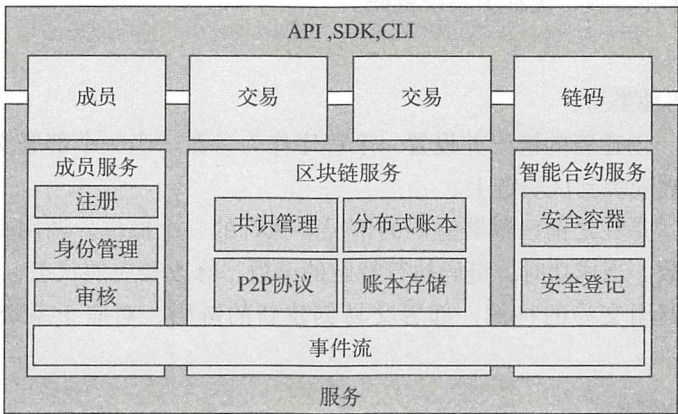


图 2-1 Fabric-v0.6 框架

如图 2-2 所示的是 Fabric-v1.0 的架构。

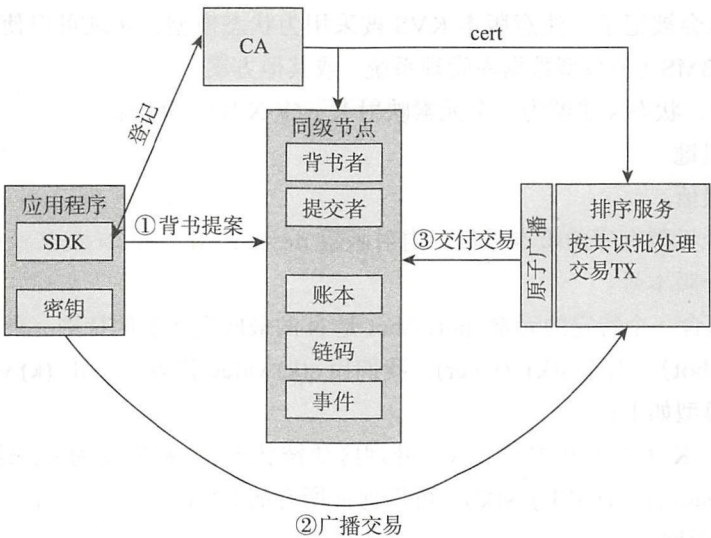


图 2-2 Fabric-v1.0 框架

Hyperledger Fabric v1 相关的架构要素为（1）系统架构，（2）交易背书的基本工作流程，（3）背书策略。

2.1.2 系统架构

区块链是一个分布式系统，由许多相互通信的节点组成。区块链运行的程序称为链码，保存状态、账本数据和执行交易。链码是核心要素，交易操作在链码上调用。交易必须被“背书”，经过背书的交易才可以提交，并对状态产生影响。有可能存在一个或多个特定的



链码用于管理功能和参数，统称为系统链码。

1. 交易

交易可以有两种类型：

❑ 部署交易。创建新的链码并设置一个程序作为参数。当一个部署交易执行成功，表明链码已被安装到区块链上。

❑ 调用交易。是在之前已部署链码的情况下执行的一个操作。调用交易引用链码提供的一个函数。当成功时，链码执行特定的函数，修改相应的状态，并返回一个输出。

部署交易是调用交易的特例，部署交易创建新的链码，对应于系统链码的一个调用交易。

2. 区块链数据结构

(1) 状态

区块链的最新状态（简称为状态）被建模为一个版本键/值存储（KVS），键的名称和值可以是任意的。整体上由运行在区块链上的链码操控，通过 KVS 操作实现。状态持续存储和状态的更新也会被记录。注意版本 KVS 被采用为状态模型，实现可以使用实际的 KVS，也可以使用 RDBMS（关系型数据库管理系统）或其他方案。

更正式地说，状态 s 建模为一个元素映射 $K \rightarrow (V \times N)$ ，其中：

❑ K 是一组键。

❑ V 是一组值。

❑ N 是一个无限有序的版本号集。内射函数 $\text{next}: N \rightarrow N$ 获取 N 的一个元素并返回下一个版本号。

V 和 N 都包含一个特定的元素 $\text{\textbackslash bot}$ ， $\text{\textbackslash bot}$ 是 N 的最底层元素的特例。最开始时所有的键都映射到 $(\text{\textbackslash bot}, \text{\textbackslash bot})$ 。对于 $s(k)=(v, \text{ver})$ ，我们用 $s(k).value$ 代表 v ，用 $s(k).version$ 代表 ver 。

KVS 操作模型如下：

❑ $\text{put}(k, v)$ 。 K 中的 k 和 V 中的 v ，处理区块链状态 s ，将它变为 s' ，这样 $s'(k)=(v, \text{next}(s(k).version))$ ，且 $s'(k')=s(k')$ ，可以保证所有的 $k' \neq k$ 。

❑ $\text{get}(k)$ 。返回 $s(k)$ 。

状态由 peer 节点保持，不是由排序节点和客户端保持。

KVS 中的键能够通过它们的名字识别它们属于哪个特定的链码。从这点上说，只有某个链码的交易可以修改属于这个链码的键。原则上，任何链码都能读取属于其他链码的键。支持跨链交易，修改属于两个或更多链码的状态是 $v1$ 的一个功能。

(2) 账本

账本提供了在系统运行过程中发生的可验证历史，它包含所有成功的状态更改（我们称之为有效交易）和不成功的状态更改尝试（我们称之为无效交易）。

账本是由排序服务构建的一个全部有序的交易哈希链块（有效的或无效的）。哈希链强



制将全部排序块置入账本，每个块包含一批全部的排序交易。

账本保存在所有 peer 节点，或者保存在排序者的一个子集中。在谈论排序时我们说的账本是排序账本，而谈论 peer 节点时我们说的账本是 peer 账本。peer 账本与排序账本的区别是，peer 节点会在本地维护一个位掩码来隔离有效交易和无效交易。

账本允许 peer 节点重演所有交易的历史和重建状态。如图 2-3 所示。

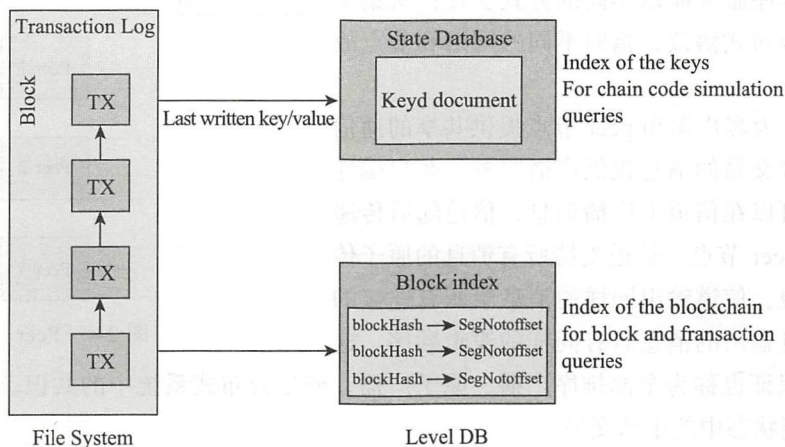


图 2-3 账本

3. 节点

node 节点是区块链的通信实体。一个 node 节点仅仅是一个逻辑函数，在这个意义上，多个不同类型的 node 节点可以运行在同一台物理服务器上。关键在于 node 节点如何在“信任域”中分组和关联控制它们的逻辑实体。

有三种类型的 node 节点。

❑ 客户端或者提交客户端：客户端提交实际交易调用到背书者，广播交易请求到排序服务节点。

❑ Peer 节点：提交交易、维持状态和账本的拷贝。此外，peer 节点可以有一个特殊的背书角色。

❑ 排序服务节点或排序者：运行通信服务实现交付保证，像原子或全序广播。

node 节点的类型接下来进行更详细地解释。

(1) 客户端

客户端代表最终用户实体。它必须连接到一个 peer 节点以便与区块链交互。客户端可以选择连接任何 peer 节点，创建并调用交易。

(2) Peer

Peer 节点以块的形式从排序服务接收有序状态更新，维持状态和账本。

Peer 节点能附加一个特殊的背书节点角色。背书节点的特殊功能是关于特殊链码，存

在于提交之前会背书一个交易。每个链码可以指定一个背书策略，引用一组背书节点。策略定义一个有效交易背书的必要和充分条件（典型的是一组背书者签名）。在部署交易的特殊情况下，安装链码（部署）背书策略是由系统链码的背书策略指定。如图 2-4 所示。

(3) Orderer

排序者产生排序服务，即一个提供交付保证的通信架构。排序服务能以不同的方式实现：从集中服务排序的分布式协议，指向不同的网络和节点故障模型。

排序服务为客户端和 peer 节点提供共享的通信信道，为包含交易的消息提供广播服务。客户端连接到信道，可以在信道上广播消息，信道随后传递消息给所有 peer 节点。信道支持所有消息的原子传递。换句话说，信道输出同样的消息给所有连接的 peer 节点并且输出的消息具有同样的逻辑顺序。这个原子通信保证也称为全部排序广播、原子广播，或是分布式系统中的共识。通信消息是包含在区块链状态中的申请交易。

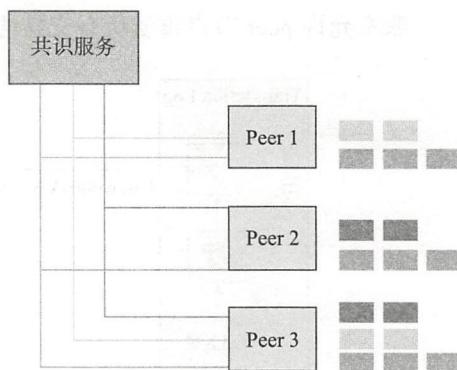


图 2-4 Peer

排序服务可以支持多个信道，类似发布 / 订阅主题消息系统。客户端能够连接到一个给定的信道，然后发送消息和获得到达的消息。信道是相互隔离的，客户端连接到一个信道时无法检测到其他信道的存在，但客户端可以连接到多个信道进行数据通讯。

peer 节点通过排序服务提供的接口连接到排序服务提供的信道。排序服务 API 包含两个基本操作（更多是异步事件）：

TODO 为获取客户端或 peer 指定的序列号（范围）的对应的块增加接口。

❑ broadcast(blob)。客户端调用此函数来广播任意消息 blob 在全信道散播。在 BFT 环境中向服务器发送一个请求时，它也被称为 request(blob)。

❑ deliver(seqno, prevhash, blob)。排序服务在 peer 节点传送带有非负整型序列号 (seqno) 和 blob 的最近哈希 (prevhash) 的消息 blob 时调用这个。换言之，它是从排序服务中产生的输出事件。deliver() 有时在发布 / 订阅系统中也称为 notify()，或在 BFT 系统中称为 commit()。

账本包含了排序服务输出的所有数据。概括地说，它是一系列 deliver(seqno, prevhash, blob) 事件，根据之前描述的 prevhash 计算形成的一个哈希链。

大多数情况下，出于效率的原因，代替输出单个交易 (blobs)，排序服务会批量输出 blobs，而且输出块是在一个单个交付事件中。在这种情况下，排序服务必须在每个块内实施和传递 blobs 的确定顺序。块内 blobs 的数量可以由排序服务实现动态选择。

接下来，为了便于说明，我们将定义排序服务的属性并解释在假定每个 deliver 事件一个 blob 的情况下，交易背书的工作流程。这些交易扩展到区块，根据上述区块内 blobs 的

确定性排序, 假定一个区块的 deliver 事件与区块内的每个 blob 对应的各自的一系列独立的 deliver 相一致。

排序服务特性

排序服务的保证(或原子广播信道)规定了广播消息的发生和交付消息之间存在什么关系。这些保证如下:

1) 安全性(一致性保证): 只要 peer 节点连接到信道足够长的时间(它们能够断开或崩溃, 但会重启和重新连接), 它们会看到交付(seqno, prevhash, blob)消息的同等序列。这意味着向所有 peer 节点输出(deliver(events))相同排序, 以及根据序列号和为相同序列号携带同等内容(blob 和 prevhash)。注意这仅是一个逻辑顺序, 在一个 peer 节点上的 deliver(seqno, prevhash, blob)是不需要与另外一个 peer 节点输出的 deliver(seqno, prevhash, blob)发生任何实时关系。换句话说, 给定一个特定的 seqno, 不会有两个正确的 peer 节点交付不同的 prehash 或 blob 值。此外, 除非一些客户节点调用了 broadcast(blob), 并且每个广播的 blob 只被调用一次, 否则不传输 blob 的值。

此外, deliver() 事件包含之前 deliver() 事件(prevhash)的数据加密哈希。当排序服务实现原子广播保证, prevhash 会从序列号为 seqno-1 的 deliver() 事件中得到参数的加密哈希。对于第一个 deliver() 事件特例, prevhash 有一个缺省值。

2) 活跃度(交付保证): 排序服务的活跃度保证由排序服务实现确定。准确的保证可以依赖于网络和节点故障模型。

原则上, 如果提交客户端没有失败, 排序服务应该保证每个连接到排序服务的正确 peer 节点最终能够传递每个提交交易。

概括地说, 排序服务保障了以下特性:

- 一致性。对于任何两个具有相同 seqno 的正确 peer 节点的事件 deliver(seqno, prevhash0, blob0) 和 deliver(seqno, prevhash1, blob1), prevhash0==prevhash1 且 blob0==blob1。
- 哈希链完整性。对于任何在正确 peer 节点中的两个事件 deliver(seqno-1, prevhash0, blob0) 和 deliver(seqno, prevhash, blob), prevhash = HASH(seqno-1||prevhash0||blob0)。
- 没有跳过。如果排序服务在正确 peer 节点 p 输出 deliver(seqno, prevhash, blob), 这样 seqno>0, 然后 p 交付事件 deliver(seqno-1, prevhash0, blob0)。
- 没有创造。任何在正确 peer 节点上的事件 deliver(seqno, prevhash, blob) 之前一定有一个 broadcast(blob) 事件在一些(可能是不同的) peer 节点上。
- 没有重复。对于任何两个事件 broadcast(blob) 和 broadcast(blob'), 当两个事件 deliver(seqno0, prevhash0, blob) 和 deliver(seqno1, prevhash1, blob') 发生在正确的节点和 blob == blob' 上时, seqno0==seqno1 且 prevhash0==prevhash1。
- 活跃性。如果正确的客户端调用事件 broadcast(blob), 那么每个正确的 peer 节点“最终”发出事件 deliver(, , blob)。

2.1.3 交易背书的基本工作流程

接下来我们概述一个交易的请求流程。如图 2-5 所示。

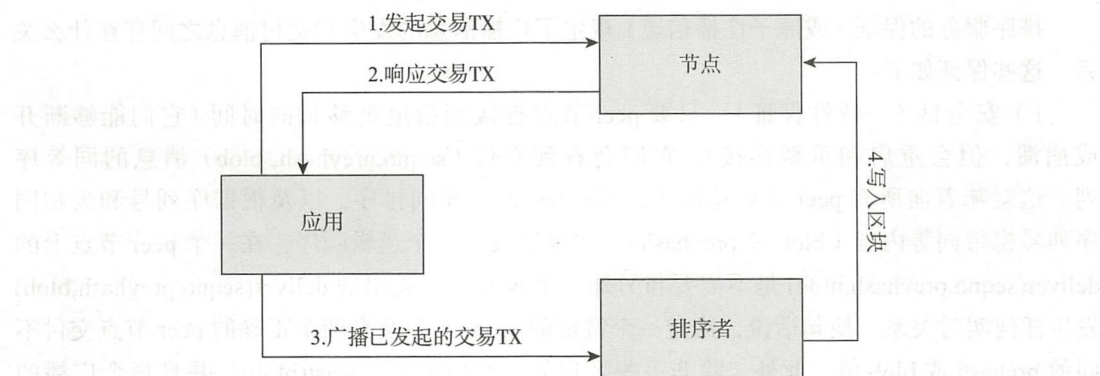


图 2-5 交易的请求流程

1. 客户端创建交易且将其发送给它选择的背书 peer 节点

调用交易，客户端发送一个 PROPOSE 消息到它选择的一组背书 peer 节点。给定 chaincodeID 的背书 peer 节点的设置由客户端通过 peer 节点实现，从背书策略知道背书 peer 节点的设置。例如，交易能被发送给所有给定 chaincodeID 的背书地者，也就是说，一些背书者能够离线，其他人可以反对或选择不为交易背书。提交客户端尝试满足背书者可用的背书策略表达。

接下来，将描述 PROPOSE 消息格式，讨论在提交客户端和背书者之间可能的互动模式。

PROPOSE 消息格式

一个 PROPOSE 消息的格式中 tx 是强制的，anchor 可选参数会在下面列出。

clientID 是提交客户端的身份，chaincodeID 引用交易相关的链码，txPayload 是提交交易自身的载体，timestamp 是由客户端维护的一个单独递增整型值，clientSig 是 tx 的其他域客户端签名。

txPayload 的细节会在调用交易和部署交易之间有所不同。

对于调用交易，txPayload 会包含两个域：

- ❑ operation 表示链码操作函数和参数。
- ❑ metadata 表示调用相关的属性。

对于部署交易，txPayload 会包含三个域：

- ❑ source 表示链码的源码。
- ❑ metadata 表示链码和应用的相关属性。
- ❑ policies 包含所有 peer 节点可访问的链码的相关策略，例如背书策略。注意背书策略在部署交易中不支持 txPayload，但部署的 txPayload 包含背书策略 ID 和它的参数。

- anchor 包含读版本依赖, 更具体地说是键 - 版本对 (即, anchor 是 $K \times N$ 的一个子集), 它捆绑或“锚” PROPOSE 请求到指定 KVS 中 key 的版本。如果客户端指定 anchor 参数, 背书者背书交易只基于读它本地 KVS 匹配 anchor 中的相应 Key 的版本号。

tx 加密哈希被所有 node 节点用做唯一的交易标识 tid (即, $\text{tid}=\text{HASH}(\text{tx})$)。客户端在内存中保存 tid, 等待背书 peer 节点的响应。

客户端决定与背书者互动的顺序。例如, 客户端通常会发送交易信息 (即, 没有 anchor 参数) 到一个单独的背书者, 背书者随后产生版本依赖 (anchor), 客户端可以在晚些时候使用这个版本依赖 (anchor) 作为它的 PROPOSE 消息参数, 发送给其他背书者。另外, 客户端能直接发送交易信息 (没有 anchor) 到它选择的所有背书者。不同的通信模式都有可能, 客户端在这方面是自由的。

2. 背书 peer 节点模拟交易和产生背书签名

在从客户端接收消息时, 背书 peer 节点 epID 首先校验客户端签名 clientSig, 然后模拟一个交易。如果客户端指定了 anchor, 那么背书 peer 节点模拟交易只基于在它本地 KVS 匹配的由 anchor 指定的版本号对应的 key 读版本号 (即下面定义的 readset)。

模拟一个交易涉及背书节点尝试执行一个交易 (txPayload), 通过调用链码到交易引用 (chaincodeID) 和背书 peer 节点来完成本地持有的状态拷贝。

作为执行的结果, 背书 peer 节点计算读版本依赖 (readset) 和状态更新 (writeset), 也在 DB 语言中称为 MVCC+postimage info。

回顾状态包含键 / 值对。所有键 / 值对实体都是版本化的, 也就是说, 每个实体包含排序版本信息, 它是在每次键的值更新时增加的。解释交易的 peer 节点记录了所有被链码访问的键 / 值对, 不管读还是写, peer 节点不会更新它的状态。更具体地说:

- 在背书节点执行一个交易前给定状态 s, 被交易读取的每个键 k, 键 / 值对 (k,s(k).version) 被添加到 readset。
- 此外, 对于每一个被交易编辑的键 k 到值 v', 键 / 值对 (k,v') 被添加到 writeset。v' 能成为新值与前值 (s(k).value) 的增量。

如果客户端在 PROPOSE 消息中指定了 anchor, 那么客户端指定的 anchor 在模拟交易时必须等于背书 peer 节点产生的 readset。

然后, peer 节点内部提交交易提案到它的逻辑部分来背书交易, 称为背书逻辑。缺省时, 一个 peer 节点的背书逻辑接受交易提案并简单签署。背书逻辑可以执行任意功能。

如果背书逻辑决定背书的一个交易, 它会发送消息到提交客户端, 其中:

- 交易提案。=tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,writeset), 其中 txContentBlob 是链码 / 交易专用信息。目的是让 txContentBlob 用作 tx 的一些陈述 (例如, $\text{txContentBlob}=\text{tx.txPayload}$)。
- epSig 是背书 peer 节点的交易提案签名。

否则，如果背书逻辑拒绝背书交易，背书者可以发送消息（TRANSACTION-INVALID, tid, REJECTED）到提交客户端。

注意背书者在这一步不能改变它的状态，在背书没有影响状态的情况下交易模拟产生状态更新。

3. 提交客户端收集交易背书并通过排序服务广播它

提交客户端一直等待直到它在（TRANSACTION-ENDORSED, tid, , ）上收集到“足够”的消息和签名来推断出交易提案已背书。

“足够”的准确数字取决于链码背书策略。如果满足了背书策略，则说明该交易已经被背书，注意此时该交易还没提交（Committed）还没提交。经过背书节点签过名的 TRANSACTION-ENDORSED 消息的集合包含了被背书过的交易。

如果提交客户端没有设法为交易提案收集背书，则放弃这个交易，稍后再试。

对于一个具有有效背书的交易，我们会使用排序服务。提交客户端使用 broadcast(blob) 调用排序服务，其中 blob=endorsement。如果客户端没有能力直接调用排序服务，它可以通过它选择的 peer 节点代理广播。这样的 peer 节点必须被客户端信任不会从背书移除任何消息或其他可能被无效的交易。注意一点，无论如何，代理 peer 节点不可能制造有效背书。

4. 排序服务向 peer 节点提交交易

当一个事件（seqno, prevhash, blob）发生并且一个 peer 节点已为所有序列号低于 seqno 的 blobs 更新状态，peer 节点执行流程如图 2-6 所示。

- ❑ 它检查 blob.endorsement 是有效的，根据的是它引用的链码（blob.tran-proposal.chaincodeID）。
- ❑ 典型的情形是，与此同时也确认依赖（blob.endorsement.tran-proposal.readset（dependencies，即版本依赖）仍没有被违背。更复杂的使用情况，背书中 tranproposal 字段可能不一样，在这种情况下，背书策略规定了状态如何演化。根据为状态更新选择的一致性（或者说“隔离度保证”，isolation guarantee）属性，版本依赖的校验可以用不同的方法实现。可串行性（Serializability）是一个默认的隔离度保证，除非 chaincode 背书策略指定一个不同的。要求在 readset 中与每个 key 相关联的版本值等于，在状态中这些 key 对应的版本值，并且拒绝那些不满足该要求的交易，这样则能提供所谓的串行性。
- ❑ 如果所有的这些检查通过，交易被视为有效或承诺。在这种情况下，peer 节点在 PeerLedger 用 1 标记交易，适用于 blob.endorsement.tran-proposal.writeset 区块链状态（如果交易提案是相同的，其他背书策略逻辑定义了函数处理 blob.endorsement）。
- ❑ 如果 blob.endorsement 背书策略验证失败，交易无效，并且 peer 节点在 PeerLedger 的位掩码用 0 标记交易。重要的是要注意无效交易不会改变状态。

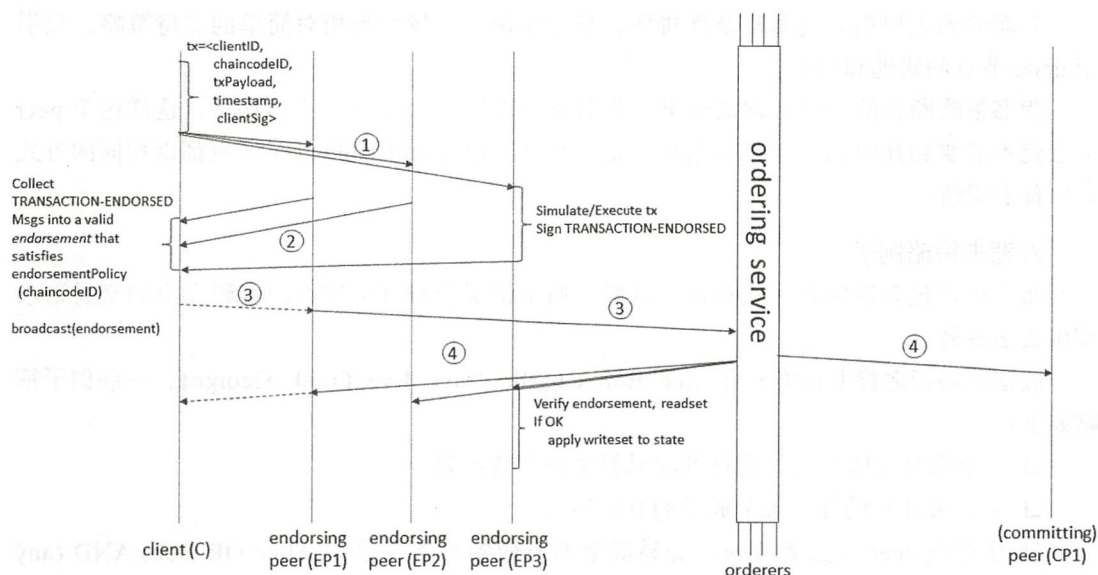


图 2-6 提交交易

2.1.4 背书策略

背书策略，是背书一个交易的条件。区块链 peer 节点有一组预先确定的背书策略，它 是被安装特定链码的部署交易引用。背书策略能参数化，这些参数能被部署交易指定。

为了保证区块链和安全特性，背书策略组应该是一组经过验证、由有限功能集合组成的策略集合，目的是确保边界的执行时间（终止）、决定性、执行和安全担保。

背书策略的动态添加（即，在链码部署时间由部署交易添加）对背书评估时间限制（终止）、决定、性能和安全保证是非常敏感的。因此，动态添加背书策略是不允许的，但未来可能会支持。

1. 针对背书策略的交易评估

交易只有经过根据背书策略的背书才会宣布有效。对于链码的调用交易首先需要 一个满足链码策略的背书，否则这个交易不会被提交。这通过在提交客户端和背书 peer 节点之间的互动发生。

正式的背书策略是以背书为基础，以及潜在的进一步评估为真假状态。对于部署交易， 获得背书的依据是系统范围策略（例如，来自系统链码）。

背书策略断言引用一定的变量。潜在可能引用的是：

- ❑ 与链码有关的钥匙或身份（在链码元数据中能发现）。例如，一组背书者。
- ❑ 链码进一步的元数据。
- ❑ endorsement 和 endorsement.tran-proposal 的元素。
- ❑ 其他更多。

上面的列表根据表现和复杂性排序，意思是说，它将会是相对简单的支持策略，只引用 node 节点的钥匙和身份。

背书策略断言的评估必须被确定。背书应当被每个 peer 节点本地评估，这样这个 peer 节点就不需要和其他 peer 节点在这件事情上交互，但所有正确的 peer 节点都以相同的方式评估背书策略。

2. 背书策略例子

断言可以包含逻辑表达式和评估真假。通常情况会对背书节点为链码发出的交易请求使用数字签名。

假定链码指定背书者集 $E=\{\text{Alice, Bob, Charlie, Dave, Eve, Frank, George}\}$ ，一些例子策略如下：

- ❑ 一个来自全体 E 成员同样的交易提案的有效签名。
- ❑ 一个来自 E 的任一单个成员的有效签名。
- ❑ 从背书 peer 节点来的同一交易提案的有效签名条件是：(Alice OR Bob) AND (any two of: Charlie, Dave, Eve, Frank, George)。
- ❑ 同一提案的有效签名需为 7 名背书者的任意 5 名。(更常用的，链码 $n>3f$ 背书者， n 名背书者有任意 $2f+1$ 个有效签名，或任意大于 $(n+f)/2$ 背书者小组有效签名。)
- ❑ 假定背书者有一个“股份”或“权重”的任务，像 $\{\text{Alice}=49, \text{Bob}=15, \text{Charlie}=15, \text{Dave}=10, \text{Eve}=7, \text{Frank}=3, \text{George}=1\}$ ，其中全部股份是 100：策略需要一组占大多数股份的有效签名（即，一组合并股份完全超过 50），例如 $\{\text{Alice, X}\}$ ，X 只要不是 George，其余谁都可以，或 $\{\text{除去 Alice 以外的所有人}\}$ 等。
- ❑ 假定前面例子中的股权条件是静态的（固定在链码的元数据中）或动态的（取决于链码的状态和在执行中的修改）。
- ❑ 交易提案 1 的有效签名来自 (Alice OR Bob)，交易提案 2 有效签名来自 (Charlie, Dave, Eve, Frank, George 中的任何两个)，其中交易提案 1 和交易提案 2 的不同只有它们的背书 peer 节点和状态更新。

如何使用这些策略取决于应用、失败或恶意背书者的恢复能力和各种其他特性。

2.1.5 证实账本和节点账本检查

1. 验证账本

维护一个账本的抽象，让其只包含有效和提交交易（例如比特币的方案），除状态和账本外，peer 节点可以维护证实账本（或 VLedger）。这是一个哈希链，来自过滤掉无效交易的账本。如图 2-7 所示。

证实账本块的生成按如下顺序。当节点账本块可能包含无效交易，即交易的背书无效或版本依赖无效时，这样的交易被 peer 节点在交易从块变为证实块之前过滤掉，每个 peer

节点自身实现这点。证实块被定义为没有无效交易的块，是经过过滤的块。这样证实块在大小上是动态的也可能是空的。

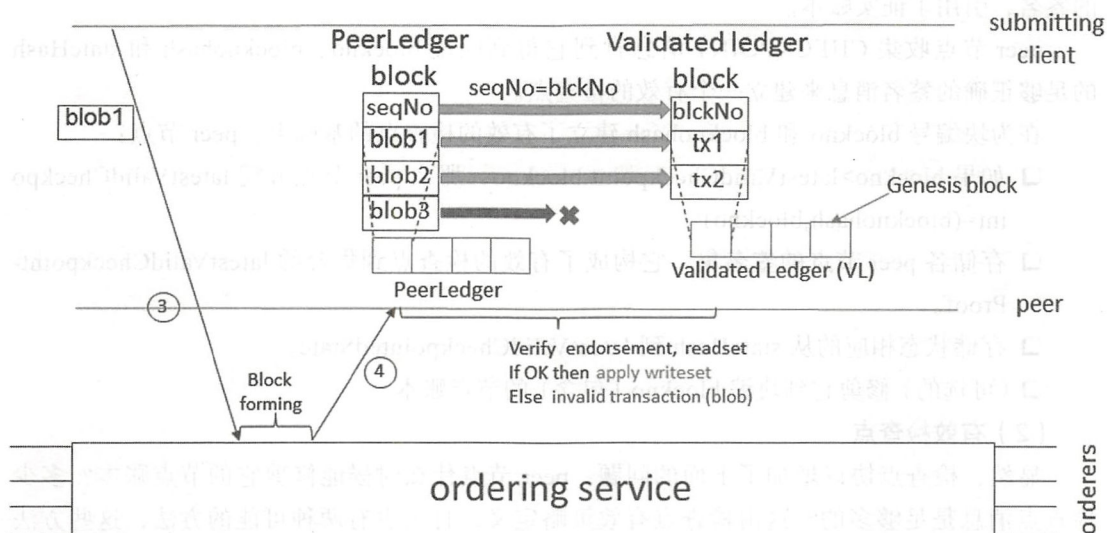


图 2-7 验证账本

证实块被每个 **peer** 节点链接在一起形成一个哈希链。更具体地，证实账本的每个块包含：

- ❑ 前证实块的哈希。
- ❑ 证实块编号。
- ❑ 从上一个证实块被计算出以来所有 **peer** 节点提交交易的排序列表，即在相应块中的有效交易列表。
- ❑ 相应块的哈希（在节点账本中），来自得出的当前证实块。

所有这些信息都被 **peer** 节点级联和哈希，产生证实账本中证实块的哈希。

2. 节点账本检查

账本包含的无效交易，没有必要永久记录。然而，一旦建立相应的证实块，**peer** 节点不能简单地丢弃节点账本块。即，在这种情况下，如果新的 **peer** 节点加入了网络，其他 **peer** 节点不能转移丢弃块到新加入的节点，也不能使新加入的 **peer** 节点承认它们的证实块。

为了便于节点账本修剪，官方文档描述了一个检查点机制。这个机制建立了证实块的有效性，贯穿节点网络，允许检查点证实块替换丢弃的节点账本块。反过来，减少了存储空间，因为没有必要存储无效交易。它也减少了新加入的 **peer** 节点重构状态的工作量。

(1) 检查点协议

检查点是由 **peer** 节点的每个 **CHK** 块周期性形成的，这里的 **CHK** 是一个可配置参数。开辟一个检查点，**peer** 节点广播（传播）给其他 **peer** 节点，其中，**blockno** 是当前块编号，

blocknohash 是各自的哈希, stateHash 是最新状态的哈希 (产生于, 例如 Merkle hash), 基于确认的块编号, peerSig 是 peer 节点对 (CHECKPOINT, blocknohash, blockno, stateHash) 的签名, 引用了证实账本。

peer 节点收集 CHECKPOINT 消息直到它得到匹配 blockno、blocknohash 和 stateHash 的足够正确的签名消息来建立一个有效的检查点。

在为块编号 blockno 和 blocknohash 建立了有效的检查点的基础上, peer 节点:

- ❑ 如果 $\text{blockno} > \text{latestValidCheckpoint.blockno}$, 那么 peer 节点分配 $\text{latestValidCheckpoint.int} = (\text{blocknohash}, \text{blockno})$ 。
- ❑ 存储各 peer 节点的签名集, 它构成了有效的检查点到集合的 latestValidCheckpoint-Proof。
- ❑ 存储状态相应的从 stateHash 到 latestValidCheckpointedState。
- ❑ (可选的) 修剪它到块编 blockno (包含) 的节点账本。

(2) 有效检查点

显然, 检查点协议增加了下面的问题: peer 节点什么时候能修剪它的节点账本? 多少检查点消息是足够多的? 这由检查点有效策略定义, 且至少有两种可能的方法, 这些方法可以相互结合。

- ❑ Local (peer-specific) checkpoint validity policy (LCVP)。给定 peer 节点 p 上的本地策略可以确定一组 peer 节点, 这一组 peer 节点是 p 信任的且它的 CHECKPOINT 消息足够建立一个有效的检查点。例如, 在 peer 节点 Alice 上的 LCVP 可以定义本地 (peer 确定) 检查点的有效性策略 (LCVP)。
- ❑ Global checkpoint validity policy (GCVP)。检查点有效策略可以确定为全局的。这类似于本地节点策略, 除非是在系统链上的规定, 而不是 peer 节点层面的规定。例如, GCVP 可以指定:
 - ❑ 每个 peer 节点可以信任一个由 11 个不同 peer 节点确认的检查点。
 - ❑ 在具体部署中每个排序者与 peer 节点配置在同一台机器上 (即, 信任域), 多达 f 个排序者可以是 (拜占庭) 错误, 每个 peer 节点可以信任一个检查点, 如果经过 f+1 个排序者配置的不同的节点确认。

2.2 Fabric 交易流程

Fabric 的正常交易流程是基于整个 Fabric 网络已经搭建完毕, 并正常运行的前提下。用户已经注册并且使用组织认证授权 (CA) 登记, 同时获得必要的加密材料来进行网络验证。链码被安装在节点上并在通道上进行实例化, 链码包含定义交易指令集合的逻辑并设置一项针对链码的背书策略, 表明哪些节点需要为交易进行背书。如图 2-8 所示。

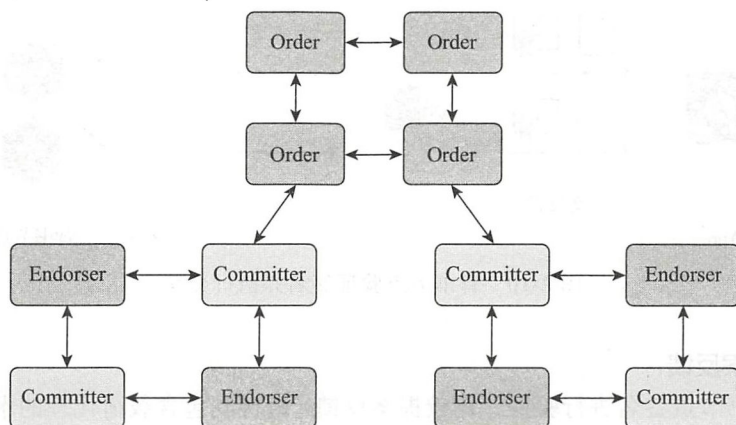


图 2-8 交易流程

1. 客户端发起交易

客户端发出交易提案请求。请求需要的是交易进行背书的目标节点。背书策略表明哪些节点必须为交易进行背书。可以结合 Fabric SDK (Node、Java、Go、Python) 开发相关客户端应用来创建提案。这项提案作为调用链码功能请求来完成数据到账本的读取或写入（即为资产写入新的键值对）。SDK 有两个作用：一个作用是将提案封装成符合 grpc 协议的 protobuf 格式的消息，另一个作用是对创建的交易提案进行签名。如图 2-9 所示。

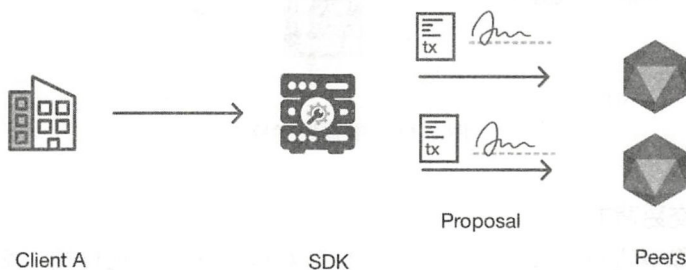


图 2-9 客户端发起交易

2. 背书节点验证签名并执行交易

背书节点使用 MSP 模块验证签名并确定请求者是否被合理授权进行提案的操作（使用每个 channel 对应的 ACL 进行验证）。背书节点以交易提案凭证为输入，基于当前状态的数据库执行来生成交易结果，输出包括反馈值、读取集合和写入集合。截至现在账本还未进行更新。这些值的集合、背书节点的签名以及是 / 否的背书声明一同作为“提案反馈”被传输回到 SDK，SDK 为应用解析 payload 以供进行业务逻辑的处理。如图 2-10 所示。

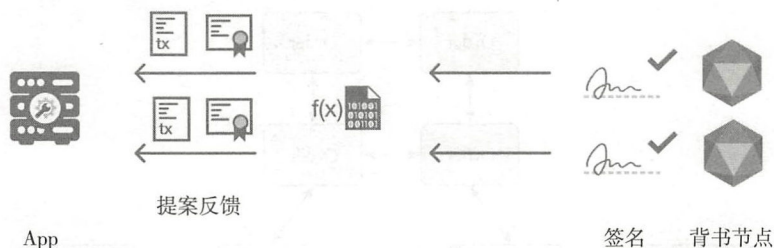


图 2-10 背书节点验证签名并执行交易

3. 审查提案反馈

应用对背书节点签名进行验证，比较提案反馈（链接到包含载荷代理的术语条款）来决定是否一致，以及指定的背书策略是否被执行（即节点 A 和 B 都进行了背书）。这种架构可以保证即使一个应用选择不进行反馈审查或者转发了没有背书的交易，背书策略依然会被节点执行并在验证提交阶段维持。如图 2-11 所示。



图 2-11 审查提案反馈

4. 客户组合交易背书

应用对交易提案进行广播，以“交易信息”对订购服务实现反馈。交易包含读/写集合、背书节点签名和通道 ID。订购服务不读取交易细节，只是从网络中所有的通道接收交易，根据每个通道按时间顺序调用，创建每个通道的交易区块。如图 2-12 所示。



图 2-12 客户组合交易背书

5. 交易验证和提交

交易区块被发布到通道中的所有节点。验证区块中的交易来确保背书策略被执行并且账本的读取集合变量没有发生变化，因为读取集合是执行交易生成的。区块中的交易被标记为有效或无效。如图 2-13 所示。

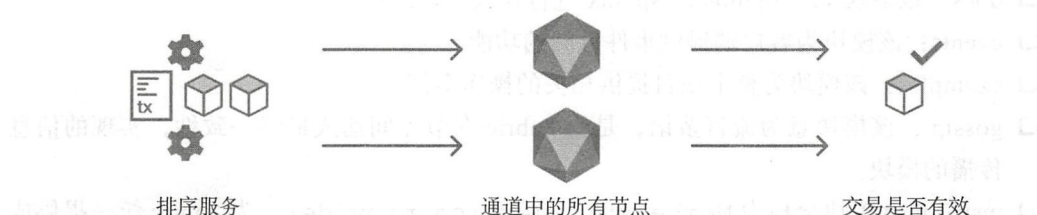


图 2-13 交易验证和提交

6. 账本更新

每个节点都把区块追加到通道的链中，对每项有效交易，写入集合被提交到当前状态的数据库。发出事务通知客户端应用，交易（调用）被永久追加到链中以及交易是有效或者无效的。如图 2-14 所示。

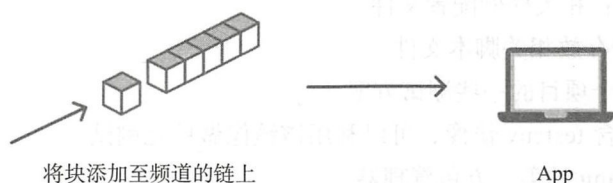


图 2-14 账本更新

2.3 Fabric 整体项目结构介绍

2.3.1 Fabric 项目结构

这里简单介绍一下 Hyperledger Fabric 项目的目录结构，以便读者能够更好地对整个项目有一个宏观的认识：

- ❑ **bccsp**：bccsp 全称是 blockchain cryptographic service provider，提供了加密标准以及算法的实现，为整个项目提供统一的加密、签名、验签服务。
- ❑ **common**：common 模块提供了通用功能以及一些通用的代码实现，包括日志、错误、工具包等，主要包括项目全局的功能性代码。
- ❑ **core**：core 模块为 Fabric 项目的核心代码模块，其中包括权限控制、chaincode 模块、commmitter、endorser、ledger、policy 等核心功能的代码实现。

- ❑ devent：主要是历史遗留原因，早期 docker 不支持 Windows 系列操作系统，使用 vagrant 运行 Ubuntu，再启动 Docker，以此来构建开发环境，但是目前各大操作系统已支持 Docker，因此不建议采用该方案。
- ❑ discovery：该模块旨在为客户端程序提供服务发现的功能。
- ❑ docs：该模块基于 Python 的 sphinx 进行在线文档的构件。
- ❑ events：该模块为客户端提供事件订阅的功能。
- ❑ examples：该模块为整个项目提供相关的操作案例。
- ❑ gossip：该模块意为流言蜚语，是为 Fabric 在节点间达成最终一致性，实现的信息传播的模块。
- ❑ msp：msp 模块全称为 Membership service provider，为 Fabric 统一提供会员服务。
- ❑ orderer：进行全局的交易排序以及切块，并推送给 peer。
- ❑ peer：包含 peer 节点的入口代码，以及命令行操作相关功能。
- ❑ proposals：存放相关提案。
- ❑ protos：存放 Protocol buffer 消息。
- ❑ release_notes：各个版本的 changelog。
- ❑ sampleconfig：相关样例配置文件。
- ❑ scripts：用于存放相关脚本文件。
- ❑ test：包含整个项目的一些测试方案。
- ❑ unit-test：包含 testenv 镜像，可以利用该镜像做单元测试。
- ❑ vendor：Golang 的第三方包管理器。
- ❑ Makefile：用于编译 Fabric。

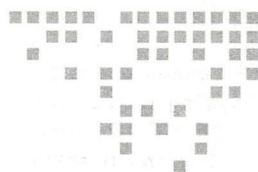
2.3.2 Fabric 源码中相关缩写的含义

下面解释的缩写是 Fabric 源码中较常见的缩写，读者可略读进行了解，或在实践中遇到时返回查看。

- ❑ MSP：Membership service provider，会员服务提供者。
- ❑ BCCSP：blockchain cryptographic service provider，区块链加密服务提供者。
- ❑ ab：atomic broadcast，原子（操作）广播。
- ❑ lsc：lifecycle(L) system(S)，chaincode (CC) 生命周期系统链码。
- ❑ Spec：Specification，规格标准，详细说明。
- ❑ KV：key-value，键 - 值。
- ❑ CDS：ChaincodeDeploymentSpec，智能合约部署规格。
- ❑ CIS：ChaincodeInvocationSpec，智能合约调用规格。
- ❑ mgmt：management，管理。

- ❑ SW: software-based, 基于软件。
- ❑ AB: AtomicBroadcast, 原子(操作)广播。
- ❑ GB: genesis block 创世纪的 block, 也就是区块链中的第一个块。
- ❑ CC 或 cc: chaincode, 链码。
- ❑ SCC 或 scc: system chaincode, 系统链码。
- ❑ csc: config system chaincode, 处理在 peer 程序端的频道配置。
- ❑ lsc: lifecycle system chaincode, 处理生命周期请求。
- ❑ esc: endorser system chaincode, 通过对交易申请的应答信息进行签名, 来提供背书功能。
- ❑ vsc: validator system chaincode, 处理交易检验, 包括检查背书策略和版本在并发时的控制。
- ❑ qsc: querier system chaincode, 提供账本查询接口。
- ❑ alg: algorithm, 算法。
- ❑ mcs: mspMessageCryptoService, 消息加密服务。
- ❑ mock: 用于单元测试的环境及数据的文件, 模拟数据环境的作用。
- ❑ Gossip: 一种分布式消息算法。
- ❑ attr: attribute, 属性。
- ❑ FsBlockStore: file system block store, 文件系统块存储。
- ❑ vdb: versioned database, 也就是状态数据库。
- ❑ RTEnv: runtime environment, 运行环境。
- ❑ pkcs11: pkcs#11, 一种公匙加密标准。
- ❑ MCS: mspMessageCryptoService, 消息加密服务。
- ❑ sa: SecurityAdvisor, 安全顾问。
- ❑ impl: implement, 用于接口或定义的实现。
- ❑ FSM: finite state machine, 有限状态机。
- ❑ FS: filesystem, 文件系统。
- ❑ blk: block, 模块。
- ❑ cli: command line interface, 命令行界面。
- ❑ CFG: FABRIC_CFG_PATH, config 的意思。
- ❑ mgr: manager, 经理。
- ❑ cpinfo: checkpoint information, 检查点信息。
- ❑ DevMode: development mode, 开发模式。
- ❑ Reg: register, 注册、登记。
- ❑ hdr: header, 头部。
- ❑ impl: implement, 实行。

- ❑ oid: ObjectIdentifier, 对象标识符。
- ❑ ou 或 OU: organizational unit, 组织单元。
- ❑ CRL: certificate revocation list, 废除证书列表。
- ❑ prop: proposal, 提案。
- ❑ ACL: Access Control List, 访问控制列表。
- ❑ rwset: read/write set, 读写集。
- ❑ tx, Tx: transaction, 交易。
- ❑ CSP: cryptographic service provider, 加密服务提供者。
- ❑ opt: option, 选项。
- ❑ opts: options, 多个选项。
- ❑ SKI: 当前证书标识, 所谓标识, 一般是对公匙进行一下 hash。
- ❑ AKI: 签署方的 SKI, 也就是签署方的公匙标识。
- ❑ HSM: Hardware Security Modules, 硬件安全模块。
- ❑ ks: KeyStore, Key 存储, 这个 key 指的是用于签名的公钥私钥。



第 3 章 Chapter 3

源码分析

本章将介绍 Fabric 中的四个模块：Logging 日志模块、Error 错误处理框架、Config 配置模块、GRPC 服务。理解日志模块有助于理解源码，理解错误处理框架能使读者独立处理代码中常见的错误，理解配置模块对具象化程序很有帮助，理解 GRPC 有助于理解如何实现客户端与服务器端的远程调用，从而为进一步理解源码打下基础。

3.1 Logging 日志模块浅析

日志指的是程序运行过程中打印到终端的日志文件，记录程序运行过程的日志。日志系统的机制是程序运行和调试必不可少的一部分，但是对于阅读源码仅仅是一种辅助，在阅读源码过程中起到了提示的作用。我们可以学习 Fabric 形成日志系统的过程，因为理解日志模块有助于提升阅读源码的流畅性。

Fabric 的日志系统主要使用了第三方包 go-logging，可在 github.com/op/go-logging 下载。

go-logging 是 Go 语言的一个基础的日志框架，支持不同的日志后端，包括系统日志、文件和内存等形式。每个 logger 实例及日志等级可以使用不同的日志后端。目前在 Github 上已经拥有 1177 个 star。

Fabric 项目很少一部分使用了 Go 语言标准库中的 log。在 go-logging 基础上，Fabric 封装出了 flogging，代码集中在 `fabric/common/flogging` 目录下，供项目全局使用。

3.1.1 go-logging 简介

logging 封装了各种打印格式，包括消息层级上的，如调试、消息、注意、警告、错误，也包括消息颜色上的，如消息是正常的绿色、错误则是醒目的红色。

基本用法如下：

```
//创建一个名字为exemplename的日志对象log
var log = logging.MustGetLogger("exemplename")
//创建一个日志输出格式对象format，确定输出格式
var format = logging.MustStringFormatter(
    `${color}%{time:15:04:05.000} %{shortfunc} ▶ %{level:.4s} %{id:03x}%{color:reset}
    %{message}` ,
)
//创建一个日志输出对象backend，也就是日志打印的位置，在此是标准错误输出
backend := logging.NewLogBackend(os.Stderr, "", 0)
//将输出格式与输出对象绑定
backendFormatter := logging.NewBackendFormatter(backend, format)
//将绑定了格式的输对象设置为日志的输出对象
//log打印会按格式输出到backendFormatter所代表的对象里，在此即是标准错误输出
logging.SetBackend(backendFormatter)
//log打印依据Info信息
log.Info("info")
//log打印一句Error信息
log.Error("err")
```

更多详细的用法，请在 go doc 或库中自行学习。可以参 <http://godoc.org/github.com/op/go-logging>，或者执行：

```
$ godoc github.com/op/go-logging
```

3.1.2 flogging

在 flogging 目录下有两个文件——grpclogger.go 和 logging.go。

grpclogger.go 用于设置 grpc 的日志，因为 grpc 默认只用 Go 语言的标准日志接口，因此将 logging 封装成 Go 语言的标准日志形式的结构 type grpclogger struct {logger *logging.Logger}，然后通过 initgrpclogger() 生成对象供 grpc 使用，从而实现让 grpc 也使用 flogging 的效果。

logging.go 文件中，自带一个名为 flogging 的日志记录者 logger，同时规定了默认的日志格式和日志等级，用 defaultFormat 和 defaultLevel 常量表示。默认的输出端是 defaultOutput，用于存放所有 Fabric 模块日志的级别映射 modules map[string]string，从类型上看其存储的日志级别都字符串化了。最后还有一个存放每个 peer 启动之时的日志级别的映射 peerStartModules map[string]string，由在每个 peer 启动完成之时调用 SetPeerStartupModulesMap() 初始化，并可通过调用 RevertToPeerStartupLevels() 恢复初始值。

3.1.3 init 函数、MustGetLogger 函数与其他函数

1. init 函数

init() 函数通过调用 Reset() 函数等，初始化了一系列默认值，如默认的输出端被设置成标准错误输出，默认输出级别则设置成 info 级别。最后调用 initgrpclogger() 初始化了

grpc 的日志对象。

2. MustGetLogger 函数

在许多不同模块的源码中，在全局的开始处都有一句类似这样的调用，如在 fabric/peer/main.go 中：`var logger = flogging.MustGetLogger("main")`。这就是调用 `MustGetLogger` 函数生成一个指定了名字的日志对象，用以记录该模块的日志，并用安全的方式（用锁的方式）将该对象记录日志的级别备案到 `modules` 中。`MustGetLogger` 函数内部依然用的是 `go-logging` 库的相应函数 `logging.MustGetLogger()` 生成的日志对象。

3. 其他函数

在 `logging.go` 中的其他函数，基本都是封装了 `go-logging` 库函数，供 Fabric 全局使用。如 `SetModuleLevel` 函数，其实就是封装了 `go-logging` 库中的 `logging.SetLevel()` 函数，以达到符合 Fabric 自己需求的目的。

3.2 Error 错误机制设计

3.2.1 总体概览

Fabric 的错误处理框架可以在 Fabric 代码仓库的 `common/errors` 目录下找到。它定义了一种新的错误类型——`CallStackError`，用于取代 Go 标准库中实现的错误类型。

一个 `CallStackError` 包含以下内容：

- ❑ **Component code**。一个生成错误信息的错误码的通用区域的组件名称。Component code 应该由 3 个大写字母组成，不允许出现数字和特殊字符。一系列 component code 被定义在 `common/errors/codes.go` 文件中。
- ❑ **Reason code**。一个用于在错误出现时定位错误原因的较短的错误码。Reason code 应该由 3 位数字组成，不允许出现字母和特殊字符。一系列 reason codes 被定义在 `common/errors/codes.go` 文件中。
- ❑ **Error code**。由冒号分隔的 component code 和 reason code 组成的错误码，例如 `MSP : 404`。
- ❑ **Error message**。描述错误信息的文本。这与提供的 `fmt.Errorf()` 和 `Errors.New()` 类似。如果一个错误被包含到当前的错误中，那么它的错误消息将被附加。
- ❑ **Callstack**。错误出现时的调用堆栈。如果一个错误被包含在当前的错误中，那么它的错误消息和调用堆栈信息会被附加到被包含的错误的上下文中。

`Callstack` 接口暴露了以下方法：

- ❑ **Error()**。返回一个带有调用堆栈的错误消息。
- ❑ **Message()**。返回一个错误消息。（不包含调用堆栈信息）。
- ❑ **GetComponentCode()**。返回由 3 个字母组成的组件代码。

- ❑ `GetReasonCode()`。返回由 3 个数字组成的错误原因代码。
- ❑ `GetErrorCode()`。返回错误代码，由 “component:reason” 组成。
- ❑ `GetStack()`。仅返回调用堆栈。
- ❑ `WrapError(error)`。将提供的错误包装进 `CallStackError`。

3.2.2 使用说明

使用新的错误处理框架来替换所有调用 `fmt.Errorf()` 或者 `Errors.new()` 的程序段。用新的错误处理框架会生成一个调用堆栈，并将其附加到错误消息中。

这个错误框架简单易用，只需要简单地调整你的代码。

首先，你需要将 `github.com/hyperleger/fabric/common/errors` 导入到使用此框架的任何文件中。

以 `core/chaincode/chaincode_support.go` 为例：

```
err = fmt.Errorf("Error starting container: %s", err)
```

对于这个错误，我们将简单地调用 `Error` 的构造函数，并传递一个组件代码、原因代码，然后是错误消息。最后，我们调用 `WrapError()` 函数，传递错误本身。

```
fmt.Errorf("Error starting container: %s", err)
```

变成

```
errors.ErrorWithCallstack("CHA", "505", "Error starting container").WrapError(err)
```

你也可以仅编写错误信息，不会有任何问题：

```
errors.ErrorWithCallstack("CHA", "505", "Error starting container: %s", err)
```

如果使用这种方法，你将能够将上一个错误消息格式化成一个新的错误，但是将失去打印调用堆栈的能力（如果包装的错误是 `CallStack`）。

另一个凸显的例子涉及了格式化错误以外的参数指令，同时仍然包含了错误，如下所示：

```
fmt.Errorf("failed to get deployment payload %s - %s", canName, err)
```

变成

```
errors.ErrorWithCallstack("CHA", "506", "Failed to get deployment payload %s",  
canName).WrapError(err)
```

3.2.3 显示错误消息

一旦使用框架创建了错误，显示错误消息将十分简单：

```
logger.Errorf(err)
```

或者

```
fmt.Println(err)
```

或者

```
fmt.Printf("%s\n",err)
```

来自 `peer/common/common.go` 的一个例子:

```
errors.Errorf("PER: %s", "404", "Error trying to connect to local peer").  
    WrapError(err)
```

将显示错误消息:

```
PER:404 - Error trying to connect to local peer  
Caused by: grpc: timed out when dialing
```

3.2.4 错误处理的一般准则

下面介绍错误处理的一般准则,以便读者在遇到常见错误时,可参考选择如何处理。

- ❑ 在努力做某种事情的时候,你应该记录错误并忽略它。
- ❑ 在为用户请求提供服务时,则应该记录错误并返回。
- ❑ 如果错误来自其他地方,可以选择包装错误。不过最好不要包装错误,让它原样返回就好。然而,对于工具函数调用的某些情况,使用 `component code` 和 `reason code` 来包装可以帮助用户在不检查调用堆栈的情况下了解真正发生错误的位置。
- ❑ 一个 `panic` 应该在同一层通过抛出内部错误代码或启动一个恢复进程来处理,而且不允许传播到其他软件包中。

3.3 Config 配置模块的设计

配置参数的读取对于一个项目是必不可少的,尤其是对 Fabric 这种参数众多的大型项目而言。因此选择一个好的配置读取框架尤为重要,这里需要一个单独的配置模块来统一管理配置参数。阅读源码对于具象化程序也很有帮助。在 `/fabric/peer/main.go` 的 `main` 函数中,除了一系列 `mainCmd` 的命令操作,还有 `viper` 进行的一系列配置操作,并通过 `err := common.InitConfig(cmdRoot)` 进行了配置的初始化。

Fabric 获取配置的途径有:环境变量、命令行参数和各种格式的配置文件。其中以配置文件为主,环境变量和命令行参数辅助,三者可以相互作用。主要的配置文件有 `core.yaml`、`orderer.yaml` 等,在 `/fabric/sampleconfig` 中有示例。主要使用的配置代码集中在 `/fabric/core/config` 下。Fabric 中三种参数配置有优先级之分,命令行参数优先级最高,环境变量次之,优先级最低的是配置文件。

3.3.1 viper 简介

Fabric 的配置系统主要运用第三方包 `viper`,可在 github.com/spf13/viper 下载。`viper` 可

以对系统环境变量、yaml/json 等格式的配置文件甚至是远程配置进行读取和设置，并可以在不重启服务的情况下动态设置新的配置项的值并使之实时生效。viper 是一个专门处理配置的解决方案，拥有眼镜蛇的称号，和命令行的库 cobra 出自同一个开发者 spf13 之手（Go 语言的开发者之一）。如图 3-1 所示即为 viper 的 logo。



图 3-1 viper 的 logo

viper 的基础用法如下：

```
//设置一个要读取的配置文件名（不包含后缀），一个viper只支持一个文件名
viper.SetConfigName("config")
//设置一个搜索配置文件的路径，viper的搜索路径可以有多个
viper.AddConfigPath("/etc/appname/")
viper.AddConfigPath(".")
//读取配置文件
viper.ReadInConfig()
//获取其中一个name项的值
viper.Get("name")
//将name的值设置为Bill
viper.Set("name", "xxx")
```

1. viper 搜索路径和文件

peer 命令对 core.yaml 的引入也是通过 viper，具体过程如下：/fabric/peer/main.go 中定义 const cmdRoot = "core"。main 函数中调用 err := common.InitConfig(cmdRoot)，该参数一路向下传递。InitConfig 函数在 /fabric/peer/common/common.go 中定义，其中调用了 config.InitViper(nil, cmdRoot) 和 viper.ReadInConfig()。InitViper 在 /fabric/core/config/config.go 中定义，接收 cmdRoot 作为参数，最终调用了 viper.SetConfigName()，也将 core 设置为了配置文件名。common.InitConfig(cmdRoot) 中的 viper.ReadInConfig() 则读取了该配置文件。

orderer 命令使用 orderer.yaml 配置文件，由 viper 引入，具体过程如下：

在 /fabric/orderer/main.go 中 main 函数调用 config.Load()/Load/fabric/orderer/localconfig/config.go 中的定义。该文件中定义了 Prefix = "ORDERER" 和 configName string，并在 init 初始化函数中将 configName 赋值为 strings.ToLower(Prefix)，即 orderer，所用的配置文件名也为 orderer。Load 函数新建了一个用于 orderer 自己的 viper，并调用了 cf.InitViper(config, configName)，其中 config 参数为新建的用于 orderer 自身的 viper，configName 为配置文件名 orderer。InitViper 在 /fabric/core/config/config.go 中定义，最终调用了 viper.SetConfigName()，也将 orderer 设置为配置文件名 Load。随后调用了 config.ReadInConfig()，读取了配置文件。

2. InitViper

上述步骤中的 peer 和 orderer 在初始化配置文件时，最终都将调用的终点指向了 /fabric/core/config/config.go 中定义的 InitViper()。下面集中分析 InitViper。

- 首先判断环境变量 FABRIC_CFG_PATH 是否有值，如果有值，则是手工定义了 FABRIC 的配置文件所在路径。参考官方文档 Getting Started 中关于手工设置 export FABRIC_CFG_PATH=\$PWD (当前目录)。
- 若没有定义该环境变量的值，则用代码添加三个路径作为搜索配置文件的路径：当前工作目录为 \$GOPATH/src/github.com/hyperledger/fabric/sampleconfig, /etc/hyperledger/fabric。
- 当前工作目录，调用 addConfigPath(v, ".") 添加，其内部调用的是 viper.AddConfigPath()。
 - \$GOPATH/src/github.com/hyperledger/fabric/sampleconfig。通过调用 AddDevConfigPath(v) 添加。AddDevConfigPath 首先调用 GetDevConfigDir(), 读取 GOPATH 路径下是否存在 src/github.com/hyperledger/fabric/sampleconfig 目录，若存在，则通过 filepath.Join 拼接 GOPATH 和 src/github.com/hyperledger/fabric/sampleconfig, 形成完整的路径并返回。AddDevConfigPath 接着调用 addConfigPath 函数，其内部调用的是 viper.AddConfigPath()。
 - /etc/hyperledger/fabric。定义了 OfficialPath = "/etc/hyperledger/fabric" 常量，如果该路径存在，则调用 addConfigPath 加入该路径。
- 调用 SetConfigName() 设置配置文件名，所指的配置文件名 configName 是由参数传递进来的。

由经由 InitViper，形成了以下 viper 配置：

FABRIC_CFG_PATH指定的路径

1. ./
2. \$GOPATH/src/github.com/hyperledger/fabric/sampleconfig
3. /etc/hyperledger/fabric

搜索的配置文件名

core —— 核心配置，供各个模块使用

orderer —— orderer配置，orderer使用

另外注意 InitViper 的第一个参数——v *viper.Viper。在 InitViper 函数中，无论是添加搜索路径（使用的是 addConfigPath 函数），还是设置要搜索的配置文件名（viper 自身的 SetConfigName 函数），都分为全局的 viper 和特定的 viper（也就是参数 v）。最终由 viper.AddConfigPath 或 viper.SetConfigName 完成的，则是全局的：由 v.AddConfigPath 或 v.SetConfigName 完成的，则是特定的。这样就可以很方便地初始化需要单独使用 viper 的模块，如 orderer 就是单独使用一条 viper，其在 /fabric/orderer/localconfig/config.go 中的 Load 函数中，config := viper.New() 新创建一个 viper，然后将此 viper 通过参数传给 InitViper，即调用 cf.InitViper(config, configName)。

3.3.2 安全文件配置

安全配置相关的代码在 `/fabric/peer/main.go` 中没有体现，而是在 `/fabric/peer/node/start.go` 中的 `serve` 函数中才初次出现。若 `grpc` 服务中使用了 TLS 网络，则需要 `.key`、`.crt`、`.ca` 文件配套文件。

在 `/fabric/peer/node/start.go` 中的 `serve` 函数中，`secureConfig, err := peer.GetSecureConfig()` 获取安全配置。

使用的安全配置结构为 `/fabric/core/comm/server.go` 中定义的 `SecureServerConfig`，用于一个 `grpc` 服务端实例。`.key`、`.crt`、`.ca` 文件所在的目录都是在 `core.yaml` 中定义的 `tls` 文件夹中，当使用 TLS 网络时，会将这些文件的数据读取到 `SecureServerConfig` 对象中。在 `GetSecureConfig()` 函数中，使用 `ioutil.ReadFile` 读取由 `/fabric/core/config/config.go` 中定义的 `config.GetPath("...")` 函数获取的 `tls` 路径下的相应文件。如 `/etc/hyperledger/fabric/tls/server.crt`。

3.3.3 命令选项配置

以 `peer start` 命令为例，在 `/fabric/peer/node/start.go` 的 `startCmd()` 函数中，`flags.BoolVarP(&chaincodeDevMode, "peer-chaincodedev", "", false, "Whether peer in chaincode development mode")` 设置了 `peer start` 命令的选项之一为 `peer-chaincodedev`，用于赋值文件中的全局变量 `chaincodeDevMode`，该变量指定了 `chaincode` 的模式。`chaincode` 的模式在 `core.yaml` 中也有定义，`chaincode.mode` 的值为 `net`，为默认选项。而当执行 `peer start` 命令时指定了选项 `peer-chaincodedev=true`，也即将 `chaincodeDevMode` 赋值为 `true`，在 `serve()` 函数中，就会使用 `viper.Set("chaincode.mode", chaincode.DevModeUserRunsChaincode)` 将 `chaincode` 的模式值设置成了 `dev`。

3.3.4 环境变量配置

在 Fabric 中，各个 `peer` 都是在容器中运行的，因此环境变量指的是各个容器中的环境变量。在各个容器的启动脚本中对容器的一些环境变量也进行了设置。在 `peer-base-no-tls.yaml` 中，如 `peer` 容器中，设置了如下环境变量：

```
- CORE_PEER_ADDRESSAUTODETECT=true
- CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
- CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=e2ecli_default
- CORE_LOGGING_LEVEL=ERROR
...
```

在 `fabric/peer/main.go` 中的开始，即对容器的环境变量进行了获取并设置：

```
//设置了环境变量前置，在此也就是peer
viper.SetEnvPrefix(cmdRoot)
//将环境变量加载进来
viper.SetEnv("CORE_PEER_ADDRESSAUTODETECT", "true")
replacer := strings.NewReplacer(".", "_")
```

```
//将环境变量中的_换成., 这样就和yaml文件的配置相匹配了。
//因为viper读取yaml文件所形成的配置项就是按层级并以.分隔的格式, 如peer.address
viper.SetEnvKeyReplacer(replacer)
```

3.4 grpc 服务

GRPC 是谷歌开发的一项多语言开源的 RPC 技术, 在 Fabric 用于实现客户端与服务器的远程调用。比如 chaincode, 客户定义了一项 rpc 服务并相应生成了客户端代码和服务端代码, 在此基础上进行业务逻辑的开发后, 分别运行服务端代码和客户端代码, 实现客户端调用服务器端服务的目的。

3.4.1 grpc 用法的 Demo

XXX.proto 文件中定义了一个 rpc 服务:

```
service Events {
    rpc Chat(stream SignedEvent) returns (stream Event) {}
}
```

1. 命令行使用 protoc 生成对应的 XXX.pb.go 源码

在 XXX.pb.go 中, Client API for Events service 处有为供客户端使用的接口定义、接口实例以及接口实例的初始化函数。Server API for Events service 处有为供服务端使用的接口定义和注册函数。如果其中某一端或同时两端为流式 RPC, 在有流的一端, 会专门为其流生成接口定义、接口实例。可以直接使用生成的实例, 也可以自己实现接口, 自定义实例。接口定义的主要方法就是 Send 和 Recv。

```
protoc --go_out=plugins=grpc:. XXX.proto
```

2. 编写客户端代码

注意, 由于目前我们关注的是 peer node start, 而其启动的基本都是后台服务端的服务, 因此本节不涉及客户端的代码。

```
//填充grpc网络链接连接选项
var opts []grpc.DialOption
opts = append(opts, grpc.WithInsecure())
//创建连接服务器端的grpc连接对象
conn, err := grpc.Dial("0.0.0.0:7051", opts...)
defer conn.Close()
//使用连接对象做参数, 利用XXX.pb.go中的初始化函数创建grpc客户端对象
client := NewEventsClient(conn)
//调用服务
client.Chat(...)
```

3. 编写服务端代码

```
//定义一个监听对象, 即服务器监听的地址
```



```
lis, err := net.Listen("tcp", ":7051")
//创建grpc服务器选项并填充
var serverOpts []grpc.ServerOption
//创建标准的grpc服务器对象
server = grpc.NewServer(serverOpts...)
//创建服务端对象, 根据XXX.pb.go中生成的接口定义, 自己实现服务端接口
type eventSever{...}
func (e *eventSever) Chat(...) {...}
es := new(eventServer)
//使用XXX.pb.go中的注册函数注册服务, 注册到grpc服务器对象上
RegisterEventsServer(server, es)
//根据监听对象启动grpc服务器对象
server.Serve(lis)
```

3.4.2 Fabric 中的 grpc 服务接口和实例

在 `/fabric/core/comm/server.go` 中, 定义了安全配置项, 以及 GRPCServer 的接口、实现和初始化函数。默认情况下在 Fabric 中是不使用 TLS 的。

1. TLS 安全配置项

TSL, 传输层安全性的缩写, 是更新、更安全的 SSL 版本。如下所示的代码, 可选择是否使用 TLS, 并配置 TLS。

```
type SecureServerConfig struct {
    //Whether or not to use TLS for communication
    UseTLS bool
    //PEM-encoded X509 public key to be used by the server for TLS communication
    //在core.yaml中指定, 读取的tls目录下server.cert文件数据存储于此
    ServerCertificate []byte
    //PEM-encoded private key to be used by the server for TLS communication
    //在core.yaml中指定, 读取的tls目录下server.key文件数据存储于此
    ServerKey []byte
    //Set of PEM-encoded X509 certificate authorities to optionally send
    //as part of the server handshake
    //在core.yaml中指定, 读取的tls目录下ca.crt文件数据存储于此
    ServerRootCAs [][]byte
    //Whether or not TLS client must present certificates for authentication
    RequireClientCert bool
    //Set of PEM-encoded X509 certificate authorities to use when verifying
    //client certificates
    ClientRootCAs [][]byte
}
```

2. GRPCServer 接口

定义了实现 GRPCServer 接口需要实现的方法, 各个方法需要完成的任务可见代码中的注释。

```
type GRPCServer interface {
    //返回GRPCServer监听的地址
```



```

Address() string
//启动下层的grpc.Server
Start() error
//停止下层的grpc.Server
Stop()
//返回GRPCServer实例对象
Server() *grpc.Server
//返回GRPCServer使用的网络监听实例对象
Listener() net.Listener
//返回grpc.Server使用的Certificate
ServerCertificate() tls.Certificate
//标识GRPCServer实例是否使用TLS
TLSEnabled() bool
//增加PEM-encoded X509 certificate authorities到
//用于验证客户端certificates的authorities列表
AppendClientRootCAs(clientRoots [][]byte) error
//用于验证客户端certificates的authorities列表中
//删除PEM-encoded X509 certificate authorities
RemoveClientRootCAs(clientRoots [][]byte) error
//基于一个PEM-encoded X509 certificate authorities列表
//设置用于验证客户端certificates的authorities列表
SetClientRootCAs(clientRoots [][]byte) error
}

```

3. GRPCServer 实现实例

根据上文所述的接口实现的实例如下：

```

type grpcServerImpl struct {
    //server指定的监听地址，地址格式：hostname:port
    address string
    //监听address的监听对象，用于处理网络请求
    listener net.Listener
    //标准的grpc服务器，通过此对象进行各种grpc服务操作
    server *grpc.Server
    //Certificate presented by the server for TLS communication
    serverCertificate tls.Certificate
    //Key used by the server for TLS communication
    serverKeyPEM []byte
    //List of certificate authorities to optionally pass to the client during
    //the TLS handshake
    serverRootCAs []tls.Certificate
    //lock to protect concurrent access to append / remove
    lock *sync.Mutex
    //Set of PEM-encoded X509 certificate authorities used to populate
    //the tlsConfig.ClientCAs indexed by subject
    clientRootCAs map[string]*x509.Certificate
    //TLS configuration used by the grpc server
    tlsConfig *tls.Config
    //Is TLS enabled?
    tlsEnabled bool
}

```



同文件中的 `NewGRPCServerFromListener` 函数是 `grpcServerImpl` 的初始化函数，其中 `tls` 相关代码使用到了 `crypto` 下的 `tls`、`x509` 工具库。

4. peer node start 启动的 grpc 服务

在 `start.go` 的 `serve` 函数中，创建的 `GRPCServer` 服务对象有两个：`peerServer` `globalEventsServer` `peer` 服务器 `peerServer`，在 `/fabric/core/peer/peer.go` 中定义。事件服务器 `globalEventsServer`，是一个全局单例，在 `/fabric/events/producer/producer.go` 中定义。

(1) peerServer

创建 `peerServer`，追溯 `serve` 函数中 `peerServer` 对象的创建，代码最终都使用了 `/fabric/core/comm/server.go` 中的 `NewGRPCServerFromListener` 函数创建了一个 `grpcServerImpl` 实例对象。

```
//在serve函数中
peerServer, err := peer.CreatePeerServer(listenAddr, secureConfig)

//在CreatePeerServer函数中
peerServer, err = comm.NewGRPCServer(listenAddress, secureConfig)

//在新GRPCServer函数中
lis, err := net.Listen("tcp", address)
NewGRPCServerFromListener(lis, secureConfig)

//在新GRPCServerFromListener函数中，最终建立grpc标准服务器并返回grpcServerImpl
grpcServer.server = grpc.NewServer(serverOpts...)
return grpcServer
```

这是我们第一次遇到 `ChaincodeSupport` 这个对象，它分为 `ChaincodeSupport` 服务原型和对应定义的 `ChaincodeSupport` 对象。从名称上就可以看出，是对 `fabric` 的 `chaincode` 提供一系列支持服务。自然，`peer` 的 `grpc` 服务中关于 `chaincode` 的操作需要这种支持服务。而 `ChaincodeSupport` 对象本身比较复杂，在 `/fabric/core/chaincode/chaincode_support.go` 中定义，提供了一个配置值成员和 `chaincode` 的运行环境，也实现了很多接口，如该处提到的服务所需的 `Register` 函数。

`ChaincodeSupport` 服务原型在 `/fabric/protos/peer/chaincode_shim.proto` 中定义，相应生成 `chaincode_shim.pb.go` 源码，在此只展示其中生成的服务端定义。

```
//服务原型
service ChaincodeSupport {
  rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {}
}

//生成服务端的接口和注册函数
type ChaincodeSupportServer interface {
  Register(ChaincodeSupport_RegisterServer) error
}
```




```

func RegisterChaincodeSupportServer(s *grpc.Server, srv ChaincodeSupportServer) {
s.RegisterService(&_ChaincodeSupport_serviceDesc, srv)
}
//生成的服务端流的接口定义、接口实例
type ChaincodeSupport_RegisterServer interface {
Send(*ChaincodeMessage) error
Recv() (*ChaincodeMessage, error)
}
grpc.ServerStream
} //接口
type chaincodeSupportRegisterServer struct {
grpc.ServerStream
} //接口实例

//注册服务:

//在serve函数中使用registerChaincodeSupport(peerServer.Server())完成注册。在该函数中:

//创建了一个ChaincodeSupport对象,基本都是读取配置值填充成员
//ChaincodeSupport对象实现了生成的服务端接口ChaincodeSupportServer中的Register方法
ccSrv := chaincode.NewChaincodeSupport(...)
//利用生成的注册函数,完成注册
pb.RegisterChaincodeSupportServer(grpcServer, ccSrv)

```

(2) globalEventsServer

事件服务器这个全局单例没有任何成员,只有一个专用初始化函数 `NewEventsServer`, 一个 `Chat` 实现。另外 `globalEventsServer` 只是一个事件服务器的代表,实际工作的是 `initializeEvents(bufferSize, timeout)` 初始化并运行的 `eventProcessor` 对象。

```

//在/fabric/events/producer/producer.go中定义
//全局单例
var globalEventsServer *EventsServer
//定义和Chat实现
type EventsServer struct {
}
func (p *EventsServer) Chat(stream pb.Events_ChatServer) error {...}
//初始化函数
func NewEventsServer(bufferSize uint, timeout int) *EventsServer {...}

```

在 `serve` 函数中,使用 `ehubGrpcServer, err := createEventHubServer(secureConfig)` 完成了对事件服务器的创建和注册, `ehubGrpcServer` 承接的就是 `globalEventsServer` 这个全局单例。

创建 globalEventsServer:

```

//在createEventHubServer中
//创建grpcServerImpl对象,其中包含了grpc标准服务器
lis, err = net.Listen("tcp", viper.GetString("peer.events.address"))
grpcServer, err := comm.NewGRPCServerFromListener(lis, secureConfig)

//创建事件服务器, NewEventsServer返回的就是globalEventsServer

```



```
ehServer := producer.NewEventsServer(
    uint(viper.GetInt("peer.events.bufferSize")),
    viper.GetInt("peer.events.timeout"))
```

事件服务原型在 `/fabric/protos/peer/events.proto` 中定义，相应生成 `events.pb.go` 源码，在此只展示其中生成的服务端定义。

注册事件服务：

```
//服务原型
service Events {
    rpc Chat(stream SignedEvent) returns (stream Event) {}
}
//生成服务端的接口和注册函数
type EventsServer interface {
    Chat(Events_ChatServer) error
}
func RegisterEventsServer(s *grpc.Server, srv EventsServer) {
    s.RegisterService(&_Events_serviceDesc, srv)
}
//生成的服务端流的接口定义、接口实例
type Events_ChatServer interface {
    Send(*Event) error
    Recv() (*SignedEvent, error)
    grpc.ServerStream
} //接口
type eventsChatServer struct {
    grpc.ServerStream
} //接口实例

//注册服务：
//还是在createEventHubServer中
//ehServer对象实现了生成的服务端接口EventsServer的Chat方法
//利用生成的注册函数，完成注册
pb.RegisterEventsServer(grpcServer.Server(), ehServer)
```

Chat 实现：

Chat 的操作很清晰，循环接收数据然后处理数据，即处理客户端的 Chat 调用，这也是 grpc 服务端需要做的。

handler, err := newEventHandler(stream)，根据服务端流接口 stream 创建一个 handler，用于处理接收客户端发送的 SignedEvent 类型数据。

in, err := stream.Recv()，接收 SignedEvent 类型的数据，这也是 grpc 双向流的标准用法。

err = handler.HandleMessage(in)，使用 handler 处理数据，HandleMessage 是实际的数据处理函数。

在 HandleMessage 函数中，客户端发送签名过的 SignedEvent 类型数据，检查有效性后，若是注册或注销事件，则注册或注销，并返回 Event 类型数据；若是其他类型的事件，则打印一条错误消息后返回。



`evt, err := validateEventMessage(msg)`, 利用 local MSP 验证数据的有效性。关于 local MSP 的具体内容将在对应主题文章中详述。

`switch evt.Event.(type) {...}`, 判断事件类型, 并对注册事件或注销事件进行注册或注销。

`if err := d.ChatStream.Send(evt);err != nil{...}`, 若是注册事件或注销事件, 执行相应操作之后返回 Event 数据给客户端, 该数据是在验证函数 `validateEventMessage` 中获取的。

启动事件服务, 在 `serve` 函数中靠后的地方, `if ehubGrpcServer != nil {go ehubGrpcServer.Start() }` 将该服务的 grpc 服务端启动起来了。`Start` 内部调用了 grpc 服务器启动的标准函数 `server.Serve(lis)`。

5. 事件实际处理者 eventProcessor

在 `eventProcessor` 的成员中:

(1) eventConsumers

按照事件类型分类的事件处理链条, 处理链 `handlerList` 接口有两种具体实现: 一般为处理链 `genericHandlerList` 和 `chaincode` 专用处理链 `chaincodeHandlerList`。在此以一般处理链为例, 其实现了对 `handlers` 的三个操作: `add`、`del`、`foreach`。其中遍历操作 `foreach` 则对 `handlers` 中的每个 `handler` 执行了由参数指定的动作。这里的 `handlers` 映射了 `handler` 与 `bool` 值, `bool` 值起到类似于开关的作用。

`handler` 在 `/fabric/events/handle.go` 中定义, 其成员 `ChatStream` 是一个在 `events.pb.go` 中生成 `Events_ChatServer` 类型的 grpc 流接口, 用于发送流数据。`handler` 挂载了一系列操作函数, 如 `register`、`HandleMessage`、`SendMessage`、`Stop`。

(2) eventChannel

`eventChannel` 为带缓存且专门处理 Event 类型数据的事件频道, 所有的事件都是通过此频道分发出去的。缓存大小由 `core.yaml` 定义为 100, 由 `initializeEvents` 的参数带进来并设置。Event 类型由 `events.proto` 定义并对应生成 `events.pb.go` 定义。

(3) timeout

频道 `eventChannel` 是满时等待的时间, 在 `core.yaml` 中设置并有释义。

我们将从其初始化函数, 也就是上文提到的 `initializeEvents` 入手, 分析事件处理者 `eventProcessor`。

`initializeEvents` 的前两句很容易理解, `if gEventProcessor != nil{...}` 只为保证 `gEventProcessor` 的单例性质, `gEventProcessor = &eventProcessor{...}` 则为 `gEventProcessor` 创建了对象实例, 分配了内存空间。而 `addInternalEventTypes()`, 实质调用了 4 次 `AddEventType`, 且为添加内部事件类型并相应的分配了这些类型各自的处理链 `handlerList`。

添加的已知的事件类型由 `/fabric/protos/peer/events.proto` 中定义, 对应生成的 `events.pb.go` 中的四种:



- ❑ EventType_BLOCK。块事件，对应 genericHandlerList。
- ❑ EventType_CHAINCODE。chaincode 事件，对应 chaincodeHandlerList。
- ❑ EventType_REGISTER。addInternalEventTypes 中有但是 AddEventType 未做处理。
- ❑ EventType_REJECTION。拒绝事件，对应 genericHandlerList start 函数。

initializeEvents 最后一句的 go gEventProcessor.start(), 意味着启动一个 goroutine 运行全局单例 gEventProcessor 的 start 函数。start 函数是一个无限循环，不断从 eventChannel 中接收数据 Event 类型数据并处理。过程如下：

- ❑ e := <-ep.eventChannel, 获取一个事件 e。
- ❑ eType := getMessageType(e), 获取事件 e 的类型 eType。
- ❑ if hl, _ = ep.eventConsumers[eType]; hl == nil {...}, 根据 eType 获取该事件类型的处理链 hl, 同时判断该类型是否存在, 若不存在则会被忽略本次事件而继续处理下一个事件。
- ❑ hl.foreach(...), 调用 hl 的 foreach 函数, foreach 遍历了 hl.handlers 中的每个 handler, 并对每个 handler 执行第二个参数指定的动作。该动作为 func(h *handler){if e.Event != nil{h.SendMessage(e)}}, 即调用每个 handler 的 SendMessage 发送事件 e。SendMessage 则是使用 handler 中自有的 ChatStream 这个生成的 grpc 流服务接口去发送事件 e: err := d.ChatStream.Send(msg)。

流程图如图 3-2 所示。

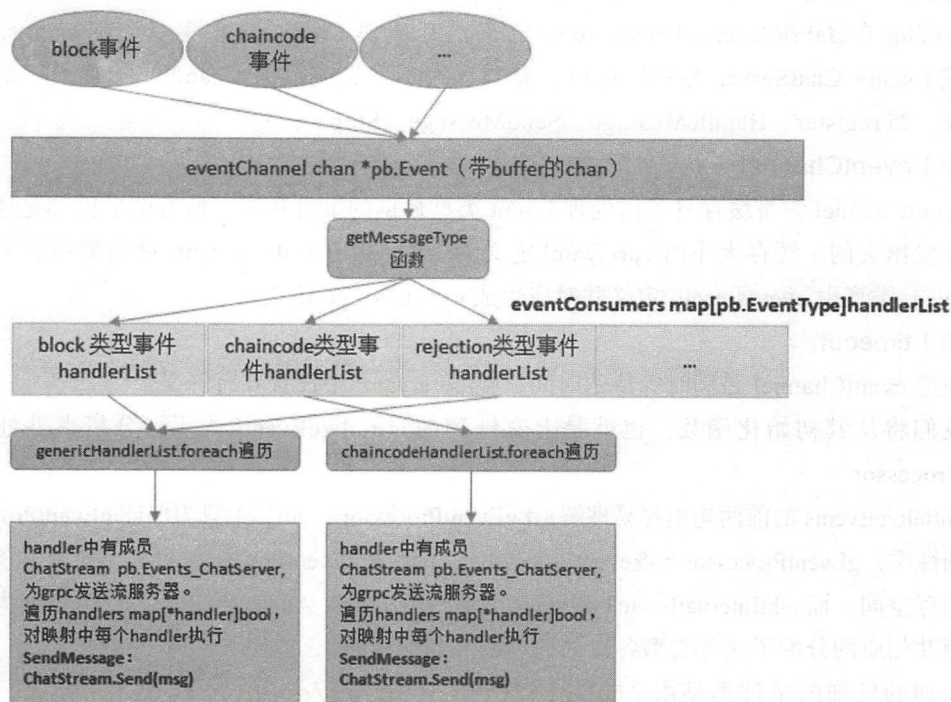


图 3-2 事件处理流程图

peer 的设计与实现

peer 节点作为 Fabric 中处理交易、存储区块的角色，承担了重要的作用，本章将讲解 peer 节点的代码实现。

4.1 CommandLine 解析

4.1.1 peer 目录结构

peer 目录结构十分清晰，只有一个 main.go 文件，其余文件夹除 common、gossip 外均为子命令集合，有 chaincode、channel、clilogging、node、version 五个，各司其职，供 main.go 整合使用。子命令文件夹中，与文件夹名称相同的 .go 文件为主要源码文件，其余的均为按功能划分的动作命令源码。以 node 目录为例，node 自身作为根命令的一个子命令，在 node.go 中实现，而 node 这个命令自身又有 start、status、stop 这三个动作去执行不同的任务，分别在对应 start.go、status.go、stop.go 中实现。注意，start、status、stop 也是子命令，是 node 这个子命令的子命令，在命令层级中负责执行动作。

具体结构关系如下所示：

- ❑ chaincode
- ❑ channel
- ❑ clilogging
- ❑ common
- ❑ gossip
- ❑ node
 - node.go



- start.go
- status.go
- stop.go
- version
- main.go

4.1.2 第三方包

在 Getting Started 中，无论是在启动 peer 容器时默认执行的命令，还是手工执行交易时在终端窗口所输入的命令，都有类似的格式，如 peer channel...、peer node...、peer chaincode...，这是一种“命令+子命令+选项”的风格。peer 命令主要依赖第三方包 github.com/spf13/cobra，由其组成基本的 peer 命令架构。所以在此简单介绍一下 cobra。

cobra 既是一个用于生成命令程序的库，也是用来生成程序和命令文件的程序（即在命令行用 cobra 命令进行一系列操作，格式化生成一些使用 cobra 框架的源代码文件，用户可在此基础上进行编程）。目前，peer 源码只将 cobra 当作一个库进行使用。cobra 的基本用法如下：

- 创建一个（根）命令对象，其原型为 Command，每个命令都是一个 Command 对象实例。创建命令对象是填充 Command 中成员的过程。需要注意的是，Command 中的成员还有很多，其中有一批字段名为 Run、RunE 形式的成员，其作用与 Run 类似，区别在于运行的时间有先有后、是否被子命令继承、是否返回错误等。

```
type Command struct {
    Use string    //命令名称字段，如你在命令行敲的是peer ...，则该字段值就是"peer"
    Short string  //短说明字段
    Long string   //详细说明字段
    Run func(cmd *Command, args []string) //该命令执行的函数
    ...
}
```

```
RootCmd := &cobra.Command{...}
```

- 如果有需要，对命令添加 flag，这一点可以简单地理解为命令选项。

```
RootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")
RootCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read from")
```

- 如果有需要，对根命令添加子命令，子命令与根命令本质是一样的，只是人为进行的级别上的区分。

```
RootCmd.AddCommand(versionCmd)
```

- 运行命令。

```
RootCmd.Execute()
```


更详细的使用方法，可在 `go doc` 或 `github.com/spf13/cobra` 上学习。

4.1.3 peer 命令结构解析

我们现在从 `peer/main.go` 文件开始解析源码，本节旨在解析 `peer` 的命令结构，因此只会涉及相关的源码，其他部分将会在后文中对应分析。如果你对 `cobra` 的用法熟悉了，就可以看懂 `main` 函数的构建。

- ❑ 首先定义了一个 `mainCmd` 命令，`var mainCmd = &cobra.Command{...}`，该命令填充了 `Use`、`PersistentPreRunE` 和 `Run` 成员。`Use` 如我们预见的那样被赋值为 `peer`，`PersistentPreRunE` 先于 `Run` 执行，都被赋值了一个匿名函数。因为 `mainCmd` 只单纯作为根命令，不实现由子命令实现的具体的交易事务，因此实现的只是 `PersistentPreRunE` 指定的检查、初始化日志系统并缓存配置的功能，以及 `Run` 指定的版本打印、命令帮助功能。
- ❑ 生成 `mainCmd` 对象的命令行标识对象 `mainFlags`，`mainFlags := mainCmd.PersistentFlags()`，也就是 `peer` 命令的选项，并对该标识对象进行了维护，增加了 `version` 和 `logging_level` 两个选项。这也对应了其在自身对象中设置 `PersistentPreRunE` 和 `Run` 的功能。
- ❑ 添加子命令，`mainCmd.AddCommand(...)`。添加的命令有 `version.Cmd()`、`node.Cmd()`、`chaincode.Cmd(nil)`、`clilogging.Cmd(nil)` 和 `channel.Cmd(nil)` 五个。`Cmd()` 是每个子命令文件中暴露出的函数，各自整合了各自的动作命令。
- ❑ 启动根命令，`mainCmd.Execute()`。启动了根命令，也就启动了其下的所有命令。

4.1.4 以 node 为例进行子命令结构解析

以 `node` 为例对子命令结构进行解析。

- ❑ 在 `node.go` 中，首先定义了一个 `node` 命令对象，`var nodeCmd = &cobra.Command{...}`
- ❑ 在 `Cmd` 函数中，添加了 `startCmd()`、`statusCmd()`、`stopCmd()` 三个函数返回的 `start`、`status`、`stop` 子命令（动作命令），分别实现在 `start.go`、`status.go`、`stop.go`，这三个命令的源码结构也是基本一致，在此仅以 `start` 和 `start.go` 为例。
- ❑ 在 `start.go` 中，首先定义了一个 `start` 命令对象，`var nodeStartCmd = &cobra.Command{...}`，其中对 `RunE` 成员赋值了一个匿名函数，函数体中执行了 `serve` 函数，这也是该命令最终会调用的函数。`serve` 函数是一个重要且复杂的函数。在每个 `peer` 容器启动后默认执行的就是 `peer node start -peer-defaultchain=false` 命令，该命令最终调用执行的就是 `serve` 函数，同时这也说明了，`serve` 函数会做很多的准备工作。

4.1.5 peer 命令结构

下述是 `Peer` 节点的所有命令：

- ❑ peer
- ❑ channel
 - join
 - create
 - fetch
 - list
- ❑ chaincode
 - instantiate
 - invoke
 - query
 - upgrade
 - package
 - install
 - signpackage
- ❑ clilogging
 - getlevel
 - setlevel
 - revertlevel
- ❑ node
 - start
 - status
 - stop
- ❑ version

4.2 Admin 及 Endorser 服务的实现

start.go 的 serve 函数、peerServer 对象在 ChaincodeSupport 服务之后，又注册了 Admin、Endorser 服务：

```
pb.RegisterAdminServer(peerServer.Server(), core.NewAdminServer())
serverEndorser := endorser.NewEndorserServer()
pb.RegisterEndorserServer(peerServer.Server(), serverEndorser)
```

4.2.1 Admin

Admin 服务实现对服务器、模块日志级别的获取和控制。服务原型定义在 /fabric/protos/peer/admin.proto 以及对应生成的 admin.pb.go 文件。核心代码在 /fabric/core/admin.go 中，该文件实现了 admin.pb.go 中的 Admin 服务端服务。

```

//proto定义的服务原型和服务状态
//在/fabric/protos/peer/admin.proto中定义
service Admin {
    rpc GetStatus(google.protobuf.Empty) returns (ServerStatus) {}
    rpc StartServer(google.protobuf.Empty) returns (ServerStatus) {}
    rpc StopServer(google.protobuf.Empty) returns (ServerStatus) {}
    ...
}
//proto使用枚举定义的服务状态
message ServerStatus {
    enum StatusCode {
        UNDEFINED = 0;
        STARTED = 1;
        STOPPED = 2;
        PAUSED = 3;
        ERROR = 4;
        UNKNOWN = 5;
    } //状态
    StatusCode status = 1;
}
//proto生成的服务端接口和注册函数
//在/fabric/protos/peer/admin.pb.go中定义
type AdminServer interface {
    GetStatus(context.Context, *google.protobuf.Empty) (*ServerStatus, error)
    StartServer(context.Context, *google.protobuf.Empty) (*ServerStatus, error)
    ...
}
func RegisterAdminServer(s *grpc.Server, srv AdminServer) {
    s.RegisterService(&_Admin_serviceDesc, srv)
}

//核心代码实现Admin服务
//在/fabric/core/admin.go中定义
type ServerAdmin struct {
}
//服务器的开启、停止、获取状态函数
func (*ServerAdmin) StartServer(context.Context, *empty.Empty) (*pb.ServerStatus, error) {
    status := &pb.ServerStatus{Status: pb.ServerStatus_STARTED}
    log.Debugf("returning status: %s", status)
    return status, nil
}
func (*ServerAdmin) StartServer(...)...{...}
func (*ServerAdmin) GetStatus(...)...{...}
func (*ServerAdmin) StartServer(...)...{...}
//模块日志的获取、设置、恢复函数
func (*ServerAdmin) GetModuleLogLevel(...)...{...}
func (*ServerAdmin) SetModuleLogLevel(...)...{...}
func (*ServerAdmin) RevertLogLevels(...)...{...}

```


4.2.2 Endorser

Endorser, 背书者。类似于对支票的背书, 表示对一种行为或权利的认可。放在 Fabric 中, 就是一个主体接收到其他某一 peer 点发送的申请消息, 通过检查该申请消息中的签名的方式, 向发送者表示支持和认可, 这样该请求或申请才可能被最终提交, 使之作用于系统。一个申请可能还需要满足一些条件, 比如必须得到一定比例数量的背书, 才算得到整个系统的认同。而这些条件, 就是指背书策略。背书者 (Endorser) 在一个交易流中充当的作用如下:

- ❑ 客户端发送一个背书申请 (SignedProposal) 到 Endorser。
- ❑ Endorser 对申请进行背书, 发送一个申请应答 (ProposalResponse) 到客户端。
- ❑ 客户端将申请应答中的背书组装到一个交易请求 (SignedTransaction) 中。

在此简单提一下交易 (Transaction) 的概念, 之后还会进行详述。交易是一个更大的概念, Fabric 的任何操作 (如 chaincode 操作, 甚至是配置系统的操作) 都被定义为一项交易。而背书只是组成交易请求数据的一个准备环节而已, 交易请求数据准备就绪后, 会被发送到 orderer。需要注意的是, 一般交易和 chaincode 交易所进行的背书过程一致, 但是背书过程交流的数据中包含的内容项有所区别。具体可参看 /fabric/protos/peer/proposal.proto 和 proposal_response.proto 中的注释和数据原型定义。

背书服务原型和实现

Endorser 服务的原型定义在 /fabric/protos/peer/peer.proto 以及对应生成的 peer.pb.go 中, 核心代码实现在 /fabric/core/endorser 下。

```
//proto中定义的服务原型
//在/fabric/protos/peer/peer.proto中定义
service Endorser {
    rpc ProcessProposal(SignedProposal) returns (ProposalResponse) {}
}
//proto生成的服务端接口和注册函数
//在/fabric/protos/peer/peer.pb.go中定义
type EndorserServer interface {
    ProcessProposal(context.Context, *SignedProposal) (*ProposalResponse, error)
}
func RegisterEndorserServer(s *grpc.Server, srv EndorserServer) {
    s.RegisterService(&_Endorser_serviceDesc, srv)
}
//核心代码实现Endorser服务
type Endorser struct {
    policyChecker policy.PolicyChecker
}
//Endorser专用初始化函数
func NewEndorserServer() pb.EndorserServer {
    e := new(Endorser)
    e.policyChecker = policy.NewPolicyChecker(
        peer.NewChannelPolicyManagerGetter(),
```

```

    mgmt.GetLocalMSP(),
    mgmt.NewLocalMSPPrincipalGetter(),
)
return e
}
//实现ProcessProposal服务
func (e *Endorser) ProcessProposal(...) {...}
func (e *Endorser) endorseProposal(...) {...}
...

```

Endorser 服务的核心实现中, 只有一个核心函数 ProcessProposal, endorser.go 中其余的函数都是相互配合供 ProcessProposal 调用, 以处理客户端发来的 SignedProposal 数据, 返回 ProposalResponse 数据, 完成最终的任务。

SignedProposal 数据结构如图 4-1 所示:

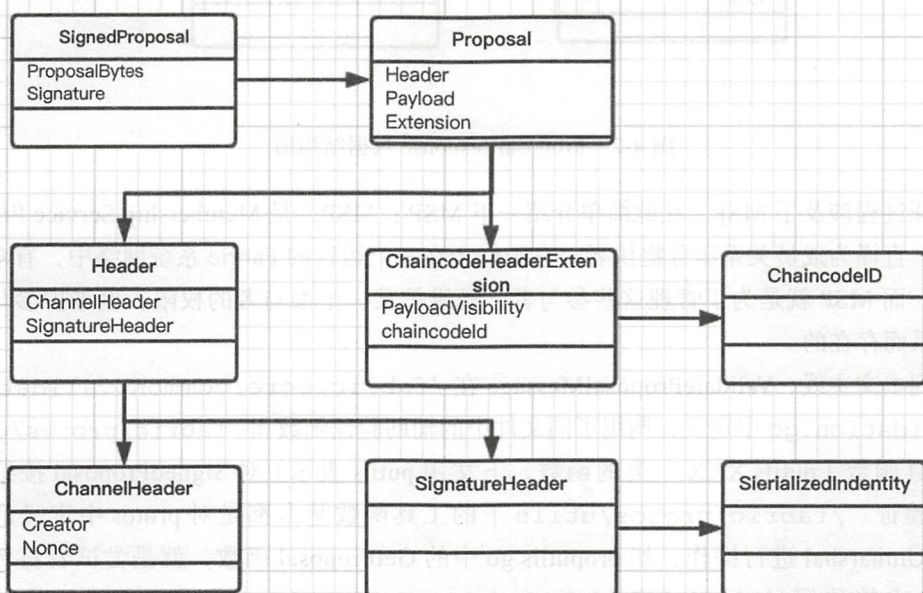


图 4-1 SignedProposal 数据结构图

ProposalResponse 数据结构如图 4-2 所示:

通过 ProcessProposal 函数可以看到 Endorser 函数是如何处理从客户端传来的 SignedProposal 数据, 以及包装何种数据到 ProposalResponse 中并将之返回。

```

//ProcessProposal 函数中调用
prop, hdr, hdrExt, err := validation.ValidateProposalMessage(signedProp)

```

首先, 第一步, ProcessProposal 函数使用 ValidateProposalMessage 对所接收的 signedProp 数据进行了验证, 并返回 signedProp 中的一些字段 Unmarshal 后的数据。

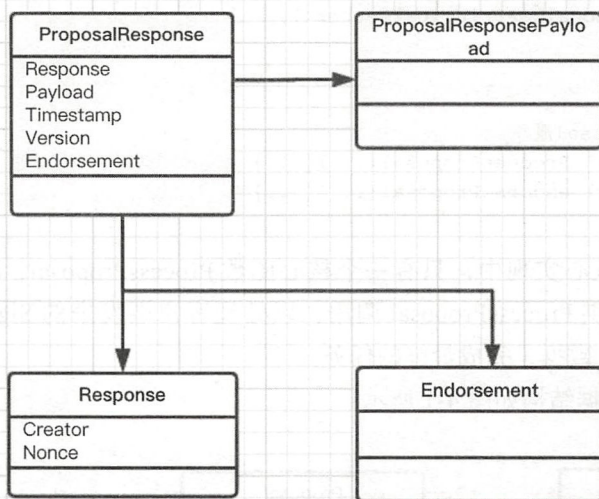


图 4-2 ProposalResponse 数据结构图

验证过程涉及了 MSP。在此简单介绍一下 MSP：MSP，是 Membership Service Provider 的缩写，直译为成员关系服务提供者。它类似于在一个运行的 Fabric 系统网络中，有众多的参与者，而 MSP 就是为了管理这些参与者，既维护某一个参与者的权限，也维护参与者之间的关系而存在的。

回到本文主线，ValidateProposalMessage 在 /fabric/core/common/validation/msgvalidation.go 中定义，调用了同文件中的辅助验证函数和 /fabric/protos/utills 下的工具函数（putils.XXX 一类的函数，下文用 putils 表示）对 SignedProposal 按结构进行逐步验证。/fabric/protos/utills 下的工具函数基本都是对 protos 中定义的数据原型的 Unmarshal 进行操作，如 proputils.go 中的 GetProposal 函数，就是尝试着将传入的 []byte 格式的数据 Unmarshal 成 /fabric/protos/peer/proposal.pb.go 中定义的 Proposal 结构数据。

```

//ValidateProposalMessage函数中调用
//与ValidateProposalMessage函数在同文件中
chdr, shdr, err := validateCommonHeader(hdr)
err = checkSignatureFromCreator(shdr.Creator, ...)
err = utills.CheckProposalTxID(...)
switch common.HeaderType(chdr.Type) {
case ...
case common.HeaderType_ENDORSER_TRANSACTION:
    chaincodeHdrExt, err := validateChaincodeProposalMessage(prop, hdr)
default:
    ...

```


validateCommonHeader 函数验证了 SignedProposal 中 ProposalBytes 的 Header，主要验证 SignatureHeader 中的 Creator 不为空、Nonce 不为空且存在，以及 ChannelHeader 中的 Type 必须是 ENDORSER_TRANSACTION、CONFIG_UPDATE、CONFIG 三者之一，并顺道返回了 Unmarshal 出来的对应的结构体数据对象。

checkSignatureFromCreator 函数利用 validateCommonHeader 顺道返回的结构体数据，验证了 SignedProposal 中的证书、背书策略，以及发起者的身份等有效性，是核心验证函数。

CheckProposalTxID 函数将 SignatureHeader 中由 Nonce 和 Creator 联合生成的 Hash 字符串值与 ChannelHeader 中的 TxId 做对比，这也同时说明了 TxId 就是 Nonce 和 Creator 联合生成的 Hash 值。这里的 Nonce 是为了防止 replay attack，在加密技术中经常用到。

validateChaincodeProposalMessage 函数简单验证了 Proposal 中的 Extension 对应的结构体 ChaincodeHeaderExtension 中的 PayloadVisibility 是否为空。该字段控制着 Proposal 的 payload 在最终的交易和在 ledger 中能用到的范围。至此，对接收的 SignedProposal 对象 signedProp 验证完毕，并返回 SignedProposal 中的 Proposal，Proposal 中的 Header 和 ChaincodeHeaderExtension 字段分别为 prop、hdr、hdrExt。对 SignedProposal 的验证内容，在 /fabric/protos/peer/proposal.proto 中定义 SignedProposal 原型时就注释了四点，可以阅读参考。

```
//ProcessProposal函数中调用
chdr, err := putils.UnmarshalChannelHeader(hdr.ChannelHeader)
shdr, err := putils.GetSignatureHeader(hdr.SignatureHeader)
if syscc.IsSysCCAndNotInvokableExternal(hdrExt.ChaincodeId.Name){...}
```

第二步，在 ProcessProposal 函数中，因为 validation.ValidateProposalMessage 没有返回，所以再次使用 putils 的函数获取 Header 的 ChannelHeader 和 SignatureHeader，分别为 chdr、shdr。然后用函数 IsSysCCAndNotInvokableExternal 验证目前处理的 SignedProposal 涉及的 chaincode 的 ID 是否是系统 chaincode，若是，验证是否能被外部调用。如果该 chaincodeID 是系统 chaincode 且不能被外部调用，则返回 true，进入 if 分支，返回相应错误。这表明，发送 SignedProposal 所在的 chaincode，若是系统 chaincode 时，要保证该系统 chaincode 允许被外界调用。在系统 chaincode 中，InvokableExternal 字段设定了是否可以被外界调用。

```
//频道ID
chainID := chdr.ChannelId
//交易ID
txid := chdr.TxId
if chainID != "" {
    lgr := peer.GetLedger(chainID)
    if _, err := lgr.GetTransactionByID(txid); err == nil { return }
    if !syscc.IsSysCC(hdrExt.ChaincodeId.Name) {
        if err = e.checkACL(signedProp, chdr, shdr, hdrExt);
        ...
    }
}
```

```

    }
  }else{
    //do nothing
  }
}

```

第三步，ValidateProposalMessage 的检查中并没有检查频道 ID 是否为空，当频道 ID 不为空时，程序根据频道 ID 调用 GetLedger 获取 peer 本地的账本 PeerLedger，然后根据交易 ID 调用账本对象自身函数，查看该交易 ID 是否已经存在于账本中，即交易的唯一性检查。接下来进行策略检查，当 chaincode 不是系统 chaincode 时，会调用背书者成员 policyChecker 的函数 checkACL，对背书者接收的 SignedProposal 是否符合所依赖频道的写者策略（writers policy of the chain）进行检查（系统 chaincode 的检查在其他地方）。成员 policyChecker 在背书者创建的时候被赋予为 policy.NewPolicyChecker(...) 的值，详细的过程写在下文的背书策略中。如果频道 ID 为空，则什么都不做。因为交易忽略了唯一性检查，没有通道 ID 的 proposal 不会影响到 ledger，也不会被提交（submitted）。没有频道 ID 的 proposal 是对照 peer 的本地 MSP 验证有效的，而不调用 ValidateProposalMessage 函数来验证 proposal 的有效性。

```

//交易模拟器接口
var txsim ledger.TxSimulator
//账本历史查询接口
var historyQueryExecutor ledger.HistoryQueryExecutor
if chainID != "" {
  //交易模拟器
  if txsim, err = e.getTxSimulator(chainID); err != nil {...}
  //账本历史查询器
  if historyQueryExecutor, err = e.getHistoryQueryExecutor(chainID); err != nil {...}
  ctx = context.WithValue(ctx, chaincode.HistoryQueryExecutorKey, historyQueryExecutor)
  defer txsim.Done()
}
//模拟交易
cd, res, simulationResult, ccevent, err := e.simulateProposal(...)

```

第四步，当频道 ID 不为空时，背书者对象使用自身的函数 e.getTxSimulator(chainID) 和 e.getHistoryQueryExecutor(chainID)，根据频道 ID 分别获取了交易模拟对象和账本历史查询对象进行模拟（simulate）交易。从这里可以看出，频道 ID(chainID) 也被作为这个频道的账本的名称，因为 peer 的账本是存在于 /fabric/core/peer/peer.go 的 chains 映射中的，映射的 key 就是频道 ID。

和第三步类似，通过频道 ID 获取账本对象，然后使用账本对象的接口 NewTxSimulator 得到交易模拟器。交易模拟器定义在 /fabric/core/ledger/kvledger/txmgmt/txmgr/lockbasedtxmgr/lockbased_tx_simulator.go 中，隶属于同目录下 lockbased_txmgr.go 中的交易管理者 LockBasedTxMgr。使用账本对象的接口 NewHistoryQueryExecutor 得

到账本历史查询器。历史查询器定义在 `fabric/core/ledger/kvledger/history/historydb/historyleveldb/historyleveldb_query_executer.go` 中，隶属于同目录下 `historyleveldb.go` 中的账本历史数据库 `historyDB`。而且，程序中调用了 `context.WithValue`，将账本历史查询器对象添加到了 `context` 中（关于 `context`，请自行学习此标准库的用法）。

得到了模拟所需的对象（交易模拟器和账本历史查询器）后，背书者使用自己的函数 `simulateProposal` 模拟交易。这里的模拟，指的是对现实交易的模拟，即 `chaincode` 中所谓的智能合约的部分被确实实地执行并产生了相应的结果集合，只不过这个过程是用数字化模拟出来的。`simulateProposal` 中使用了背书者对象的 `checkEscscAndVscsc`、`getCDSFromLSCC`、`callChaincode` 三个内调函数和交易模拟器的 `GetTxSimulationResults` 完成模拟任务。

1) `checkEscscAndVscsc` 在目前版本里直接返回 `nil`，自带 `TODO` 标签。

2) `getCDSFromLSCC`，若前面获得的交易模拟器不为空，则将其也加入 `context`。然后开始调用 `chaincode` 关于执行的代码，这里是从 `LSCC` 中获取指定名字的 `chaincode` 的数据。`chaincode` 关于执行的代码在 `/fabric/core/chaincode` 中 `chaincodeexec.go` 和 `exectransaction.go` 中，最终完成核心任务的是 `exectransaction.go` 中的 `Execute` 函数，其中使用了 `ChaincodeSupport` 服务（服务支持各个 `peer` 之间的通信交流，这也就是所谓垫片中的地位，核心逻辑由主题代码实现，而与各个 `peer` 之间通信去实现主题代码的功能，则用该服务支撑）。

3) `callChaincode`，真正执行了 `chaincode` 并返回 `HTTP` 状态应答和执行的 `chaincode` 事件。这里也调用 `chaincode` 关于执行的代码，`HTTP` 状态应答原型定义在 `/fabric/protos/commom/common.proto` 中。

4) `GetTxSimulationResults`，在 `lockbased_tx_simulator.go` 中定义，获取执行 `chaincode` 的读写集合。

至此，模拟交易函数 `simulateProposal` 执行完毕，返回 `chaincode` 数据，执行 `chaincode` 的应答信息，模拟结果集合以及 `chaincode` 的执行事件，供下一步使用。这一步中，要注意当频道 `ID` 为空时，交易模拟器即为空，之后的相关操作也会有所区别。

```
var pResp *pb.ProposalResponse
if chainID == "" {
    pResp = &pb.ProposalResponse{Response: res}
}else{
    pResp, err = e.endorseProposal(...)
}
pResp.Response.Payload = res.Payload
```

第五步，背书者对象使用自身函数 `endorseProposal` 对模拟交易进行背书，并得到交易申请应答数据 `pResp *pb.ProposalResponse`。当频道 `ID` 为空时，简单地将第四步所得的

应答信息赋予 Response 即返回，当频道 ID 不为空时，则使用上一步所返回的数据进行交易的背书。endorseProposal 中主要调用的也是背书者的内调函数 callChaincode，其上是为其准备的数据，其后是根据背书返回的数组组装申请应答信息 ProposalResponse。模拟交易和背书都调用了 callChaincode 而实现了不同的功能，主要起分别作用的是传入它的倒数第三个参数，该参数是 chaincode 的执行详细说明书 ChaincodeInvocationSpec，说明书不同的内容能指导 chaincode 执行的代码实现不同的功能。如背书功能，使用的说明书中 ChaincodeID 指定的就是系统 chaincode 中用于背书的 escc，而 chaincode 关于执行 chaincode 的代码也就根据所给定的 ChaincodeID 找到指定的 chaincode 执行。

4.2.3 频道中的策略检查器

在前文的 Endorser 中存在一个成员 policyChecker policy.PolicyChecker，该成员同样也在 Handler 对象、lsc 对象、qsc 对象中出现。如下代码定义了一个策略检查器：

```
policyChecker = policy.NewPolicyChecker(
    peer.NewChannelPolicyManagerGetter(),
    mgmt.GetLocalMSP(),
    mgmt.NewLocalMSPPrincipalGetter(),
)
```

我们通过探寻这个检查器的作用和用法，可以了解到频道中的各种策略。

策略相关的代码集中在 /fabric/core/policy 和 policyprovider、/fabric/common/cauthdsl 和 policies、/fabric/protos/common/policies.proto 和对应生成的 policies.pb.go 中。

/fabric/protos/common/policies.proto: ImplicitMetaPolicy 的原型定义。

/fabric/common/policies: 定义了 Policy 接口、Manager 接口和其实现 ManagerImpl，定义了频道策略管理者获取器 ChannelPolicyManagerGetter 接口（其现在在 /fabric/core/peer/peer.go 中的 channelPolicyManagerGetter）。对应定义了 ImplicitMetaPolicy（也是一种类型的策略）。

fabric/common/cauthdsl: 在 policy.go 中实现了 Policy 接口，定义和实现了策略对象提供者 provider，在 policyparser.go 中实现了原始字符串策略（如“OR(‘A.member’，AND(‘B.member’，‘C.member’))”）的解析（FromString 函数），cauthdsl.go 中实现了生成指定策略的评估函数，即策略的 Evaluate 接口（compile 函数），cauthdsl_builder.go 中则定义了用于生成各种所需结构的函数。

/fabric/core/policy/policy.go: 定义了 PolicyChecker 的接口和其实现 policyChecker，定义了策略检查器工厂 PolicyCheckerFactory 接口。

/fabric/core/policyprovider/provider.go: 实现了 PolicyCheckerFactory 接口 defaultFactory 并初始化了一个实例对象。

策略接口定义如下：

```
//策略接口, 在/fabric/common/policies/policy.go中
type Policy interface {
    //对比SignedData中的签名是否满足SignedData中策略
    Evaluate(signatureSet []*cb.SignedData) error
}

//错误或拒绝情况下的策略实现
type rejectPolicy string
func (rp rejectPolicy) Evaluate(signedData []*cb.SignedData) error {
    return fmt.Errorf("No such policy type: %s", rp)
}

//策略的实现1, 在fabric/common/cauthdsl/policy.go中
type policy struct {
    evaluator func([]*cb.SignedData, []bool) bool
}
func (p *policy) Evaluate(signatureSet []*cb.SignedData) error { ... }

//策略的实现2, 在/fabric/common/policiesimplicitmeta.go中
type implicitMetaPolicy struct {
    conf      *cb.ImplicitMetaPolicy
    threshold int
    subPolicies []Policy
}
func (imp *implicitMetaPolicy) initialize(config *policyConfig) { ... }
func (imp *implicitMetaPolicy) Evaluate(...) error { ... }
```

策略检查器的定义如下:

```
//策略检查器接口
type PolicyChecker interface {
    CheckPolicy(...) error
    CheckPolicyBySignedData(...) error
    CheckPolicyNoChannel(...) error
}

//策略检查器实现
type policyChecker struct {
    //频道策略管理者获取器接口, 在/fabric/peer/peer.go中实现
    channelPolicyManagerGetter policies.ChannelPolicyManagerGetter
    //本地MSP或MSPManager, 在/fabric/msp中定义和实现
    localMSP msp.IdentityDeserializer
    //MSP主角获取器, 在/fabric/msp/mgmt/principal.go中定义和实现
    principalGetter mgmt.MSPPrincipalGetter
}
func (p *policyChecker) CheckPolicy(...) { ... }
func (p *policyChecker) CheckPolicyNoChannel(...) { ... }
func (p *policyChecker) CheckPolicyBySignedData() {
    ...
    //获取策略管理者
    policyManager, _ := p.channelPolicyManagerGetter.Manager(channelID)
    //根据策略名获取策略对象, 在/fabric/common/policies/policy.go中定义和实现
    policy, _ := policyManager.GetPolicy(policyName)
}
```

```
// 评定策略
err := policy.Evaluate(sd)
...
```

至此，我们可以了解到频道中策略的存储位置。在策略检查器的三个接口中存在内部相互调用，最典型的的就是 `CheckPolicyBySignedData` 接口，该接口中用三句代码完成了策略评定。据此就可以很容易看出策略其实是存储在策略管理者对象 `policyManager` 的 `config` 成员中，策略对象的名字是其所在的路径字符串，这些路径字符串在 `/fabric/common/policies/policy.go` 中开始的部分用常量定义，进而我们就可以看到有哪些类型的策略：`ChannelReaders`、`ChannelWriters`、`ChannelApplicationReaders` 等，对应的就是频道的读者策略、频道的写者策略、频道应用的读者策略等。根据 `GetPolicy` 的过程，我们可以看出策略管理者是怎么编排其所管理的策略对象，管理者与管理者存在父子关系，呈现的管理形式也和目录结构类似。

其次是如何验证策略的问题，不同的策略有不同的验证方法，所以策略检查器 `policyChecker` 有三个接口。一般直接调用获取的策略对象的接口 `Evaluate` 进行评定，而在 `CheckPolicyNoChannel` 中则使用 `MSP` 对象（即 `MSP` 或 `MSPManager`）的 `Verify` 接口进行评定。

最后是验证了何种问题，根据背书者 `Endorser` 使用 `policyChecker` 的地方，一般在 `checkACL` 的内调函数中。而 `checkACL` 是在进行模拟交易 `simulateProposal` 之前被调用的（在 `ProcessProposal` 函数中），这也就是说，`policyChecker` 在背书过程中所做的就是检查客户端发来的请求数据是否合法，也就是检查客户端的签名，也可以说是根据频道的写者策略判断客户端在该频道中是否有权利发送请求。因此，不要把这个策略检查器与验证是否满足背书策略的那个检查（这个是发生在模拟交易后的，由 `Evaluate` 完成）混淆。

4.3 Committer 的机制

`Committer` 负责在接受交易结果前再次检查合法性，接受合法交易对账本的修改，并写入区块链结构。

4.3.1 committer.go 分析

`committer.go` 分析如下：

1. 声明了 `Committer` 接口。
2. `CommitWithPvtData` 将区块和私有数据写入账本。
3. `GetPvtDataAndBlockByNum` 使用给定的私有数据检索块。
4. `GetPvtDataByNum` 从给定区块中返回私有数据切片，并根据过滤器指示要检索私有数据的集合和名称空间的映射。

5. LedgerHeight 获取最近的区块序列号。
6. GetBlocks 根据参数提供的序列号获取区块。
7. Close 关闭提交服务。

```
type Committer interface {
    CommitWithPvtData(blockAndPvtData *ledger.BlockAndPvtData) error
    GetPvtDataAndBlockByNum(seqNum uint64) (*ledger.BlockAndPvtData, error)
    GetPvtDataByNum(blockNum uint64, filter ledger.PvtNsCollFilter) ([]*ledger.
        TxPvtData, error)
    LedgerHeight() (uint64, error)
    GetBlocks(blockSeqs []uint64) []*common.Block
    Close()
}
```

4.3.2 committer_impl.go 分析

committer_impl.go 分析如下:

1. PeerLedgerSupport 接口主要是为了抽象出 ledger.PeerLedger 接口, 去实现 LedgerCommitter 所需的 API。

```
type PeerLedgerSupport interface {
    GetPvtDataAndBlockByNum(blockNum uint64, filter ledger.PvtNsCollFilter)
        (*ledger.BlockAndPvtData, error)
    GetPvtDataByNum(blockNum uint64, filter ledger.PvtNsCollFilter) ([]*ledger.
        TxPvtData, error)
    CommitWithPvtData(blockAndPvtdata *ledger.BlockAndPvtData) error
    GetBlockchainInfo() (*common.BlockchainInfo, error)
    GetBlockByNumber(blockNumber uint64) (*common.Block, error)
    Close()
}
```

2. LedgerCommitter 结构体是 Committer 接口的实现, 它保持对账本的引用来提交区块和检索链信息。

```
type LedgerCommitter struct {
    PeerLedgerSupport
    eventer ConfigBlockEventer
}
```

3. ConfigBlockEventer 回调函数原型在到达新配置更新块时定义动作。

```
type ConfigBlockEventer func(block *common.Block) error
```

4. NewLedgerCommitter 是一个工厂函数, 用于创建提交者的一个实例, 它通过验证传入传入块, 并将它们提交到分类账中。

```
func NewLedgerCommitter(ledger PeerLedgerSupport) *LedgerCommitter {
    return NewLedgerCommitterReactive(ledger, func(_ *common.Block) error { return
        nil })
}
```

5. `NewLedgerCommitter` 是一个工厂函数，用于创建提交者的一个实例，它通过验证传入传入块，并将它们提交到分类账中。

```
func NewLedgerCommitter(ledger PeerLedgerSupport) *LedgerCommitter {
    return NewLedgerCommitterReactive(ledger, func(_ *common.Block) error { return
        nil })
}
```

6. `NewLedgerCommitterReactive` 是一个工厂函数，用于创建与 `NewLedgerCommitter` 相同的提交者实例，同时还提供一个选项来指定在新的配置块到达和提交事件时调用的回调。

```
func NewLedgerCommitterReactive(ledger PeerLedgerSupport, eventer ConfigBlock
    Eventer) *LedgerCommitter {
    return &LedgerCommitter{PeerLedgerSupport: ledger, eventer: eventer}
}
```

7. `preCommit` 负责验证块并根据其内容进行更新。

```
func (lc *LedgerCommitter) preCommit(block *common.Block) error {
    // Updating CSCC with new configuration block
    if utils.IsConfigBlock(block) {
        logger.Debug("Received configuration update, calling CSCC ConfigUpdate")
        if err := lc.eventer(block); err != nil {
            return errors.WithMessage(err, "could not update CSCC with new
                configuration update")
        }
    }
    return nil
}
```

8. `CommitWithPvtData` 以私有数据自动提交块。

```
func (lc *LedgerCommitter) CommitWithPvtData(blockAndPvtData *ledger.BlockAndPvt
    Data) error {
    // Do validation and whatever needed before
    // committing new block
    if err := lc.preCommit(blockAndPvtData.Block); err != nil {
        return err
    }

    // Committing new block
    if err := lc.PeerLedgerSupport.CommitWithPvtData(blockAndPvtData); err != nil
    {
        return err
    }

    // post commit actions, such as event publishing
    lc.postCommit(blockAndPvtData.Block)

    return nil
}
```

9. GetPvtDataAndBlockByNum 通过给定的私有数据检索块序列号。

```
func (lc *LedgerCommitter) GetPvtDataAndBlockByNum(seqNum uint64) (*ledger.
    BlockAndPvtData, error) {
    return lc.PeerLedgerSupport.GetPvtDataAndBlockByNum(seqNum, nil)
}
```

10. 一旦提交到分类账本，则通过 postcommit 发布事件或处理其他任务。

```
func (lc *LedgerCommitter) postCommit(block *common.Block) {
    // create/send block events *after* the block has been committed
    bevent, fbevent, channelID, err := producer.CreateBlockEvents(block)
    if err != nil {
        logger.Errorf("Channel [%s] Error processing block events for block number
            [%d]: %+v", channelID, block.Header.Number, err)
    } else {
        if err := producer.Send(bevent); err != nil {
            logger.Errorf("Channel [%s] Error sending block event for block
                number [%d]: %+v", channelID, block.Header.Number, err)
        }
        if err := producer.Send(fbevent); err != nil {
            logger.Errorf("Channel [%s] Error sending filtered block event for
                block number [%d]: %+v", channelID, block.Header.Number, err)
        }
    }
}
```

11. LedgerHeight 返回最近提交的块顺序号。

```
func (lc *LedgerCommitter) LedgerHeight() (uint64, error) {
    var info *common.BlockchainInfo
    var err error
    if info, err = lc.GetBlockchainInfo(); err != nil {
        logger.Errorf("Cannot get blockchain info, %s", info)
        return uint64(0), err
    }
    return info.Height, nil
}
```

12. GetBlocks 用于检索切片中提供的序号的块。

```
func (lc *LedgerCommitter) GetBlocks(blockSeqs []uint64) []*common.Block {
    var blocks []*common.Block

    for _, seqNum := range blockSeqs {
        if blk, err := lc.GetBlockByNumber(seqNum); err != nil {
            logger.Errorf("Not able to acquire block num %d, from the ledger
                skipping...", seqNum)
            continue
        } else {
            logger.Debug("Appending next block with seqNum = ", seqNum, " to the
                resulting set")
        }
    }
}
```



```

        blocks = append(blocks, blk)
    }
}

return blocks
}

```

4.3.3 validator.go 分析

validator.go 的分析如下：

该部分源码从 Support 接口声明部分开始。

1. 声明了 Support 接口，提供了评估 VSCC 所需的全部内容。
2. Acquire 实现信号量的获取语义。
3. Release 实现了信号量释放语义。
4. Ledger 返回与此验证程序相关联的账本。
5. MSPManager 返回此频道的 MSP 管理器。
6. Apply 尝试应用 configtx 以成为新配置。
7. GetMSPIDs 返回已在通道中定义的应用程序 MSP 的 ID。
8. Capabilities 定义此频道的应用程序部分的功能。

```

type Support interface {
    Acquire(ctx context.Context, n int64) error
    Release(n int64)
    Ledger() ledger.PeerLedger
    MSPManager() msp.MSPManager
    Apply(configtx *common.ConfigEnvelope) error
    GetMSPIDs(cid string) []string
    Capabilities() channelconfig.ApplicationCapabilities
}

```

1. 声明了 Validator 接口，定义 API 以验证块事务并返回位数组掩码，指示未通过验证的无效事务。

```

type Validator interface {
    Validate(block *common.Block) error
}

```

2. txValidator 结构体是 Validator 接口的实现，继续参考账本以启用 tx 仿真和执行 vscc。

```

type txValidator struct {
    support Support
    vscc     vsccValidator
}

```

3. NewTxValidator 创建新的事务验证器。

```
func NewTxValidator(support Support) Validator {
    // Encapsulates interface implementation
    return &txValidator{
        support: support,
        vscc:    newVSCCValidator(support)}
}
```

(1) Validate 执行块的验证。并行执行块中每个事务的验证方法如下：提交程序线程在 goroutine 中启动 tx 验证函数（使用信号量来限制并验证 goroutine 的数量）。该线程然后从结果通道中读取验证结果（在完成 goroutine 的订购者中）。goroutines 执行块中 txs 的验证并将验证结果排入结果通道。

(2) 为了保持验证方法简单，提交者线程将块中的所有事务排入队列，然后继续读取结果。

(3) 为了使并行验证正常工作，重要的是验证功能不会改变系统的状态。此处验证执行的顺序很重要，我们不得不求助于顺序验证（或者一些锁定）。目前就是这样做的，因为影响状态的函数是收到交易并验证交易，但这必须保证交易独立在区块中。如果这个条件改变了，代码也要改变。

```
func (v *txValidator) Validate(block *common.Block) error
```

4. generateCCKey 为特定通道中的 chaincode 生成唯一标识符。

```
func (v *txValidator) generateCCKey(ccName, chainID string) string
```

5. invalidTXsForUpgradeCC 无效所有由于 chaincode 升级 txs 而应该被废止的 txs。

```
func (v *txValidator) invalidTXsForUpgradeCC(txsChaincodeNames map[int]*sysccprovider.ChaincodeInstance, txsUpgradedChaincodes map[int]*sysccprovider.ChaincodeInstance, txsfltr ledgerUtil.TxValidationFlags)
```

4.3.4 vscc_validator.go 分析

vscc_validator.go 的分析如下：

1. 声明了 vsccValidator 接口，专用接口用来解耦 tx validator 和 vscc execution；以提高 txValidator 的可测性。

```
type vsccValidator interface {
    VSCCValidateTx(payload *common.Payload, envBytes []byte) (error, peer.TxValidationCode)
}
```

2. vsccValidator 实现，用于调用 vscc chaincode 并验证块事务。

```
type vsccValidatorImpl struct {
    support      Support
    ccprovider   ccprovider.ChaincodeProvider
    sccprovider  sysccprovider.SystemChaincodeProvider
}
```

3. newVSCCValidator 创建新的 vscc 验证器。

```
func newVSCCValidator(support Support) *vsccValidatorImpl
```

4. VSCCValidateTx 为事务执行 vscc 验证。

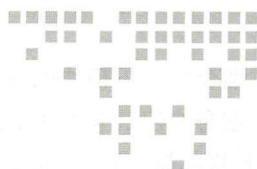
```
func (v *vsccValidatorImpl) VSCCValidateTx(payload *common.Payload, envBytes []byte) (error, peer.TxValidationCode)
```

5. GetInfoForValidate 从 lsc 获取 tx、vscc 和 policy 的 ChaincodeInstance (包含最新版本)。

```
func (v *vsccValidatorImpl) GetInfoForValidate(chdr *common.ChannelHeader, ccID string) (*sysccprovider.ChaincodeInstance, *sysccprovider.ChaincodeInstance, []byte, error)
```

6. txWritesToNamespace 确认提供 NsRwSet 执行账本写入。

```
func (v *vsccValidatorImpl) txWritesToNamespace(ns *rwsetutil.NsRwSet) bool
```

order 的设计与实现

orderer 为 Fabric 的排序节点，为所有的客户端提供统一的交易排序及打包服务，并将区块分发到所有的 leader 节点。

5.1 orderer 内部机制窥探

orderer 服务是先于所有 peer 节点启动起来的单独的节点。并且 orderer 服务与 gossip 服务密切相关。order 只做 broadcast 和 deliver。orderer 节点与各个 peer 节点通过 grpc 连接，orderer 将所有 peer 节点通过 broadcast 客户端发来的消息（Envelope 格式，比如 peer 部署后的数据）按照配置的大小依次封装到一个个 block 中并写入 orderer 自己的账本中，然后供各个 peer 节点的 gossip 服务通过 deliver 客户端来消费这个账本中的数据进行自身节点账本的同步。数据流向：peer 节点→grpc→Server→broadcast/deliver→broadcastSupport/deliverSupport→multichain→chainSupport→kafka/solo→chain→BlockCutter→chainSupport.CreateNextBlock→chainSupport.WriteBlock→ledger 账本→kafka→peer 节点。

5.1.1 kingpin

orderer 服务的命令行使用了第三方库 kingpin（v2 版本），地址为 [gopkg.in/alecthomas/kingpin.v2](https://github.com/alecthomas/kingpin)。kingpin 本身比较简单，而且 orderer 也只使用了 kingpin 最表面的一层，只有两个子命令，start 和 version，且没有任何 flag 或参数。start 是默认命令，即 docker-compose-base.yaml 中启动 orderer 节点容器时所执行的命令。当前版本中，docker-compose-base.yaml 在 examples/e2e_cli/base 下，且执行的 command 是 orderer，由于 start 是默认命令，所以执行 orderer 相当于执行了 orderer start 命令。

5.1.2 模块

下面我们详细介绍一下模块的各个组成部分。

- ❑ **Server**。服务自身，包括 broadcast 和 deliver。有以下三种类型的服务，分别用于不同情况下的部署：1) Solo ordering service(Solo)，针对开发测试环境的一种服务，特点是简单，不支持 consensus，不够高效且不可测试。2) Kafka-based ordering service，卡夫卡，一种经典的发布-订阅服务。orderer 中使用了第三方库 github.com/Shopify/sarama 创建客户端，以连接 kafka。在 fabric Getting Started 操作下载的镜像中，有一个 kafka 镜像以作为服务端处理消息（还有一个 zookeeper 镜像辅助 kafka）。在产品级别的项目部署中，这种服务是当前最好的选择，有很高的吞吐量，非常高效，但是不支持容错。3) PBFT ordering service，拜占庭容错，当前处于开发之中，这个模块是服务主体，包含多个子模块，如 cutter、kafka、过滤器等。
- ❑ **Ledger**。orderer 服务必须允许客户端能够在排序过的块流中进行查询，因此 orderer 服务需要一个支持账本。有以下三种类型的账本，分别用于不同情况下的部署：1) File ledger，文本账本，针对产品级别的项目部署使用，默认选择此类账本（下文也只讨论这类账本）。账本直接存储在文件系统上，通过轻量级 LevelDB 数据库以 Index 进行索引，以供客户端高效地检索指定的块。2) RAM ledger，内存账本，可配置用于存储的内存的大小，即一切数据都存于内存中，针对测试，不容错，重启后将重置。3) JSON ledger，JSON 账本，用 JSON 文本存储账本数据，也是用来针对测试环境的，特点在于简单明了，且可以容错，重启后不会重置。
- ❑ **Profiling**。监控 orderer 服务的模块，可供通过 http（如浏览器）来监控 orderer 的情况。

5.1.3 配置

配置是 orderer 服务启动过程中所必需的，直接影响了 orderer 的服务方式。主要涉及 orderer.yaml、configtx.yaml 两个配置文件。

- ❑ **orderer.yaml**。用于 orderer 服务自身。orderer.yaml 的数据使用被加载于 main.go 中 main 函数中的 `conf := config.Load()`，形成了 localconfig/config.go 中的 TopLevel 对象 conf (common/configtx/tool/localconfig/config.go 中也有对应的一套 TopLevel)，过程较为复杂，多个结构体一级一级地嵌套，但较方便的地方是取出来具体的某一项配置值的变量与 orderer.yaml 中对应的配置 key 一样，如要取出 orderer.yaml 中 GenesisFile 项的值，则使用 `conf.General.GenesisFile` 即可，且 config.go 中有一个默认的 TopLevel 对象 defaults 作为默认配置。
- ❑ **configtx.yaml**。主要供 orderer 生成 genesisblock 作为 orderer 使用的链的第一块 block。在 main.go 中，initializeBootstrapChannel 函数中生成 genesisblock 时，`genesisBlock = provisional.New(genesisconfig.Load(conf.General.GenesisProfile)).GenesisBlock()` 这个函数相当于 Getting Started 使

用 configtxgen 手动生成的 genesis.block, 使用何种方式生成 genesisblock 由 orderer.yaml 中的 GenesisMethod 项和启动 orderer 容器时所给的环境变量, 即 docker-compose-base.yaml 中 ORDERER_GENERAL_GENESISMETHOD 项所决定, 后者优先权应该大于前者, configtx.yaml 被 genesisconfig.Load(...) 加载, 形成的 ConfigEnvelope 被写入到 genesisblock 中。这个过程比较复杂, 经历了多重结构的嵌套。基本的数据结构嵌套变化流程是: common/configtx/tool/localconfig/config.go 中的 TopLevel→provisional/provisional.go 中的 bootstrapper (ConfigGroup)→common/configtx/template.go 中的 Template→Envelope (ConfigEnvelope)→Block (genesisblock)。而具体哪些数据写入了 genesisblock, 可以使用 test 函数将这个 ConfigEnvelope 打印出来。multichain/manager.go 中的 multiLedger 对象是 orderer 服务直接使用的总管, 所以初始化这个对象可以执行到 orderer 服务的各个子模块, 自然也包含生成 genesisblock 这一步, 所以在 manager_test.go 的 TestManagerImpl 函数中, 有初始化这个对象, 可以进行此测试。打印点设置在 common/genesis/genesis.go 的 Block(...) 中的 err = proto.Unmarshal(configEnv.ConfigUpdate, configUpdate) 下面, 使用 fmt.Printf("\033[43;36mconfigUpdate:%v\n\033[0m\n", configUpdate) 以黄底绿字的形式打印。写入 genesisblock 的配置数据主要是 configtx.yaml 中 orderer 部分的配置数据。

5.1.4 模块的初始化

1. Server

□ gprc 服务。orderer 所基于的 gprc 服务 AtomicBroadcast 原型在 protos/orderer/ab.proto 中定义, 对应生成的默认的客户端、服务端对象在 ab.pb.go 中, 其中 orderer 所操作的服务端又单独实现在 orderer/server.go 中, peer 点使用的客户端则基本沿用默认生成的客户端。server.go 中的服务端对象实例 server 在 main.go 的 main() 中由 server := NewServer(manager, signer) 生成, 使用 ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server) 进行注册, 随后由 grpcServer.Start() 启动。server 没有实际功能, 其两个服务 Broadcast、Deliver 直接对应交由两个成员 handlerImpl(orderer/common/broadcast/broadcast.go) 和 deliverServer(orderer/common/deliver/deliver.go) 的 Handle 处理。客户端的初始化, 如 peer chaincode 和 peer channel 各自会使用到命令工厂 ChaincodeCmdFactory/ChannelCmdFactory 中使用到的 BroadcastClient 客户端, 且都会随着 peer 节点的初始化而初始化。再如 peer 节点的 gossip 服务, 所使用的 Deliver 客户端 (core/deliverservice/deliverclient.go 中的 DefaultABCFactory) 也是默认生成的客户端, 会随着 gossip 模块的初始化而初始化。

- ❑ multiLedger 总管。在 `orderer/multichain/manager.go` 中定义，是 `orderer` 直接使用的“总管家”，由其支配各个子模块。`multiLedger` 包装 `chain` 集合、`consenters` 集合、签名工具、账本生成工具和系统 `chain`。在 `main.go` 的 `main()` 中，`manager := initializeMultiChainManager(conf, signer)` 完成了 `multiLedger` 的初始化。初始化完成时，所有包含的子对象也相应被初始化，所有 `orderer` 中现存的 `chain` 也启动起来。这里的 `chain` 既可以指账本本身，还可以指包含了账本的 `chainSupport`，还可以指具体的处理消息的流程（如 `orderer/solo` 和 `orderer/kafka` 下各自实现的 `chain` 所执行的 `Enqueue`）。而 `multiLedger` 是多条链的“总管家”。
- ❑ `chainSupport`。接口和实现均在 `orderer/multichain/chainsupport.go` 中定义，如同 `chaincode_support` 对 `chaincode` 一样，属于尽力对 `chain` 操作提供支持的对象。既包含账本本身，也包含了账本用到的各种工具对象，如分割工具 `cutter`、过滤工具 `filter`、签名工具 `signer`，以及最新配置在 `chain` 中的位置信息（`lastConfig` 的值代表当前链中最新配置所在的 `block` 的编号，`lastConfigSeq` 的值则代表当前链中最新配置消息自身的编号）等。`chainSupport` 也实现了一些用于支持各个工具的小接口。一些流程切换、数据流向的转变等工作，都交给了 `chainSupport` 来做。`chainSupport` 也会做一些其他的工作，比如 A 工具操作中需要调用 B 工具，则 A 包含的往往不是 B 本身，而是 `chainSupport`，借 `chainSupport` 来使用 B。
- ❑ `consenter`。分为 `solo` 和 `kafka` 两种类型，用于序列化生产（即各个 `peer` 点传送来的 `Envelope`）出来的消息。`solo` 在 `orderer/solo` 中实现，`kafka` 在 `orderer/kafka` 中实现，两者也都是起到引导和配置的作用来供 `chainSupport` 使用，真正负责的是 `solo/kafka` 封装进来的 `chain`（`orderer/solo/consensus.go`）和 `chainImp`（`orderer/kafka/chain.go`）。`solo` 的 `chain` 是 `for` 循环 + `select-case` + `chan` 组成的简单处理流程。`kafka` 的 `chainImp` 则是一套使用第三方库 `github.com/Shopify/sarama` 实现的连接 `kafka` 服务端进行消息的发布订阅的方案。由于 `solo` 和 `kafka` 不具备容错性，所以不能称之为 `consenter`。
- ❑ `chain` 的启动。这里的 `chain` 指的是处理消息的线程。在 `orderer/multichain/manager.go` 中，当 `NewManagerImpl` 初始化完毕一个 `chainSupport` 后，都会执行 `chain.start()` 启动背后的 `chain` 处理消息的流程。以 `kafka` 为例，`chain.start()` 调用的是 `orderer/kafka/chain.go` 中 `chainImpl` 的 `Start()`，进而 `go startThread(chain)` 启动了一个新线程，在 `startThread()` 中：1) `chain.producer, err = setupProducerForChannel(...)` 创建了 `kafka` 的生产者。2) `chain.parentConsumer, err = setupParentConsumerForChannel(...)` 创建了消费者。3) `chain.channelConsumer, err = setupChannelConsumerForChannel(...)` 分区消费者。4) `close(chain.startChan)`，`chain.errorChan = make(chan struct{})` 分别开启处理 `peer` 节点通过 `Broadcast`，`Deliver` 发来的请求的开关（只有开启 `orderer` 才会开始处理 `peer` 发来的

请求)。5) `chain.processMessagesToBlocks()` 启动了 for 循环接收处理消息过程, 该过程是用来接收 kafka 服务端 (在 orderer 中就是 kafka 容器) 序列化后返回给 kafka 客户端 (在 orderer 中就是 sarama) 的消息流, 然后依次分类处理。在 `processMessagesToBlocks()` 中: 1) 正常的 kafka 生产出来的消息, 会从 `chain.channelConsumer.Messages()` 这个通道中出来, 根据不同的 kafka 消息类型, 在 switch-case 中分别使用 `processConnect`、`processTimeToCut`、`KafkaMessage-Regular` 函数进行处理。

❑ `filter/committer`。过滤器/执行器, 有几种类型, 分散在 orderer 目录下, 几种 filter 可以组装到一起形成一个过滤集合, orderer 服务用此过滤集合 (`orderer/common/filter/filter.go` 中的 `RuleSet`) 中各个 filter 的 `Apply` 函数对收到的消息进行过滤, 比如有些类型的消息的一些字段不能为空, 大小必须在配置的范围之内等。和 filter 对应的是一个提交对象 `committer`, 即一个消息通过了 filter 的 `Apply`, 会返回一个 `committer`, 这个 `committer` 会在该消息写入账本之前执行 `Commit` 操作。同时, `committer` 的 `Isolated` 的返回值标识着一条消息是否要单独成块, 一些比较特殊的消息可能在不满足一定大小的情况下也要单独作为一个 block, 比如 `gensisblock`。一个 filter 集合, 就是一个合同。而一个 `Envelope` 来到 orderer 中之后, 会根据这个合同一条条地进行条款对照, 如果符合, 执行一个对应的 `committer` 工具, 以使 `Envelope` 能够拿到这个 `committer` 工具进一步完成一些事情。`filter/committer` 接口在 `orderer/common/filter` 下定义, 同时定义了三种基本的 `Apply` 过滤的结果: `Accept` 表示接受, `Reject` 表示拒绝, `Forward` 表示当前 filter 需要进一步验证。filter 有如下类型: 1) `emptyRejectRule`, 验证不能为空的 filter, 在 `orderer/common/filter/filter.go` 中定义。2) `sigFilter`, 验证签名的 filter, 在 `orderer/common/sigfilter/sigfilter.go` 中定义。这里会使用公共签名策略 (`common/policies` 中定义) 来验证进入 orderer 的 `Envelope` 中所携带的签名是否满足要求。3) `sizeFilter`, 验证大小的 filter, 在 `orderer/common/sigfilter/sigfilter.go` 中定义。在 `configtx.yaml` 中 `Orderer` 配置中有许多关于大小多少的配置项, 将在这个 filter 中验证。4) `systemChainFilter`, 系统链 filter, 在 `orderer/multichain/systemchain.go` 中定义, 作用在于验证一个 `Envelope` 是否是一个用于升级 orderer 配置的 `ConfigUpdate` 类型消息, 若是, 则会提供一个包含 `configtx` 工具的 `committer`, 在该 `Envelope` 写入账本前, 执行 `committer`, 新建一条新配置的链。5) `configtxfilter`, 在 `orderer/common/configtxfilter/filter.go` 中定义, 作用与 `systemChainFilter` 类似, 检测 `Envelope` 是否是一个 `HeaderType_CONFIG` 类型的消息, 然后返回一个包含有 `configManager` 工具的 `committer` 供之后对消息做进一步操作。

❑ `blockcutter`。块分割工具, 用于分割 block, 具体为 `orderer/common/blockcutter/blockcutter.go` 中定义的 `receiver`。类似于工厂流水线的自动打包机, 一条条的消息

数据在流水线上被传送到 cutter 处, cutter 按照 configtx.yaml 中的配置 (主要是大小限制), 把一条条消息打包成一批 (一箱) 消息, 同时返回整理好的这批消息对应的 committer 集合, 至此 cutter 的任务完成。每一批消息被当作一个 block, 执行完对应的 committer 集合后被写入账本。在 configtx.yaml 中关于 Orderer 的配置项中, BatchSize 部分规定了 block 的大小: MaxMessageCount (10) 指定了 block 中最多存储的消息数量, AbsoluteMaxBytes (10 MB) 指定了 block 最大的字节数, PreferredMaxBytes (512 KB) 指定了一条消息最优的最大字节数 (blockcutter 处理消息的过程中会努力使每一批消息尽量保持在这个值上)。根据这三个值, cutter 在工作时 (具体指 blockcutter.go 中的 Ordered 函数): 1) 若一个 Envelope 的数据大小 (Payload+ 签名) 大于 PreferredMaxBytes, 无论当前缓存如何, 立即 Cut。2) 若一个 Envelope 被要求单纯存储在一个 block (即该消息对应的 committer 的 Isolated() 返回为 true), 要立即 Cut。3) 若一个 Envelope 的大小加上 blockcutter 已有的消息的大小之和大于 PreferredMaxBytes, 要立即 Cut。4) 若 blockcutter 当前缓存的消息数量大于 MaxMessageCount, 要立即 Cut。5) 还有一个比较特殊的 Cut, 由 configtx.yaml 中 BatchTimeout 配置项 (默认 2s) 控制, 当时间超过此值, chain 启动的处理消息的流程中主动触发的 Cut。Cut 所做的工作, 就是将当前缓存的消息和 committer 返回供 blockcutter 与当前处理的 Envelope 打包成一批或两批消息, 然后清空缓存信息。在上述需要 Cut 的情况中, 只有 1) 2) 会产生两批消息, 且先是旧的消息 (即 blockcutter 中之前缓存的消息) 为一批, 后是当前处理的最新的消息为一批。receiver 中, filters 是一个 RuleSet, 定义了过滤条件集合, 均来自于 orderer/multichain/chainsupport.go 中的 createStandardFilters 或 createSystemChainFilters (后者只是比前者多了一个 systemChainFilter 过滤对象, 如 4) 所示); pendingBatch 是一个 Envelope 数组, 用来缓存 Envelope 消息; pendingBatchSizeBytes 用来记录这些缓存的消息的大小, pendingCommitters 是一个 Committer 数组, 用来缓存每个 Envelope 对应的 committer。

2. ledger

这里只讨论 file 形式的账本。orderer 中使用的账本同 peer 节点中使用的一样, 包含账本生成者 BlockStoreProvider 和账本自身 BlockStore。只不过被稍微包装了一些, BlockStore 被包装进了 fileLedger (orderer/ledger/file/imp.go 中), 而 fileLedger 又是 ReadWriter 接口 (orderer/ledger/ledger.go 中) 的实现。ReadWriter 由 Reader 和 Writer 组成, 也就是说, 这可能是会拓展的部分, 即以后在 orderer 中可能会实现只读账本、只写账本、读写账本的分类。fileLedger 是随配置工具 configResources 一同封装到 ledgerResources (orderer/multichain/manager.go) 中, 供 chainSupport 使用。

与 fileLedger 相配合的是一个同文件中的 fileLedgerIterator 迭代器, 由 fileLedger 的 Iterator(startPosition) 生成, 传入一个链上 (实际是账本中) 查询 block 开始迭代的位置赋值

给迭代器的成员 `blockNumber`，比如 `startPosition` 为 0（即 `SeekPosition_Oldest`），则说明从链的第一块 `block` 开始遍历迭代，再比如 `startPosition` 为 `SeekPosition_Newest`，则说明从链的现存最新的一块 `block` 开始遍历迭代，其余的值均算为 `SeekPosition_Specified` 类型的位置，即任意指定链上的一个 `block` 块。任何查询 `fileLedger` 的行为，都是通过循环调用这个迭代器的 `Next()` 进行的，每调用一次，`blockNumber` 自增 1。`fileLedgerIterator` 迭代器的一个特征是 `Next()` 会在迭代到还没有写入到账本 `block`（即当前账本最新的 `block` 的接下来将要有的一个 `block`）时阻塞等待，一直等待到该 `block` 被写入到账本中。这个阻塞控制涉及两个 `chan`，`impl.go` 中的 `closedChan` 和 `fileLedger` 的成员 `signal`。下面详述这个阻塞控制：

- 当迭代器当前遍历到的 `block` 序列号小于等于账本上当前最新的 `block` 序列号时（即 `ledger.Height()-1`），在 `Next()` 中，会进入 `if i.blockNumber < i.ledger.Height()` 分支（根据 `orderer` 对账本的操作，可知 `ledger.Height()` 返回的是链上将要添加的下一个 `block` 的序列号），查询后返回。而当迭代器遍历到 `ledger.Height()` 时，此时 `i.blockNumber == i.ledger.Height()`，`Next()` 会进入 `<i.ledger.signal` 等待。当一个新的 `block` 要被写入账本，则会调用 `fileLedger` 的 `Append(block)`，在 `Append(block)` 中，把 `block` 写入账本成功之后，会调用一下 `close(fl.signal)`（此时 `Next()` 的等待会结束再次进入 `if i.blockNumber < i.ledger.Height()` 分支取 `block`），然后紧接 `fl.signal = make(chan struct{})` 再给 `signal` 创建一个 `chan`（此时若再调用 `Next()`，仍会再次进入 `<i.ledger.signal` 等待）。

- `fileLedgerIterator` 迭代器的 `ReadyChan()` 根据当前迭代器的实际情况进行操作。情况 1：当 `i.blockNumber > i.ledger.Height()-1` 时，说明迭代器已经遍历到要查询账本的下一块还没有写入的 `block` 了，则返回 `signal`；情况 2：相反，则直接返回 `closedChan`。这个控制会在 `Deliver` 服务处理客户端索要 `block` 时用到，即在 `orderer/common/deliver/deliver.go` 的 `Handle(...)` 中，正常情况会进入 `if seekInfo.Behavior == ab.SeekInfo_BLOCK_UNTIL_READY` 分支进行 `select-case` 选择，此时 `case <-erroredChan`：除了关闭 `Deliver` 时会来消息，`erroredChan` 不会再来消息，只会等待 `case <-cursor.ReadyChan()`。若发生情况 1，这里将等待 `signal`，直到在 `Append(block)` 中写入新的 `block` 后 `close(fl.signal)` 一下程序才会继续前进，在下文会调用到迭代器的 `cursor.Next()`；若发生情况 2，由于 `closedChan` 是关闭的，程序将继续执行。

5.2 kafka 排序服务机制讲解

`kafka` 是一个分布式的消息队列，用于提供全局唯一的消息队列，以保证所有的消费者都能消费统一的消息。

- ❑ Producers/Consumers (生产者 / 消费者), 是最基本的概念, 指生产 / 消费消息的人 (程序、节点)。在 Fabric 中, 生产者就是各个 peer 节点, 如 peer chaincode 的一些操作生产出来一些消息发送到 orderer, 再由 orderer 送至 kafka; 消费者分为直接消费者和最终消费者, 直接消费者就是 orderer 中的 ledger, kafka 出版的消息都被直接写入了 orderer 的 ledger 之中。最终消费者是各个 peer 节点, 各个 peer 节点通过 grpc 再从 orderer 的 ledger 中取出写入的数据来同步自身的 ledger。
- ❑ Topic, 主题。生产者生产消息后, 按照主题发布消息, kafka 将处理的消息按内容主题进行分类, 到了 fabric 中的 orderer 服务中, topic 是按照 chainID 来分的, 即一个 chain 对应一个主题, 一条 chain 的 orderer 所生产的消息都会发布到同一个对应主题中。
- ❑ Partition, 分区。每个 Topic 消息可以分为多个分区进行存储。kafka 是基于文件存储的, 因此可以直接把一个分区视作一个独立的文件 (消息多, 自然要分开存在多个文件中), 接收到的消息直接会被按顺序追加到文件尾部。到了 Fabric 中的 orderer 服务中, 一个主题 (即一个 chain) 只对应一个分区。
- ❑ offset, 偏移。消息存储在每个分区中, 都会有一个 offset (类似 C 语言的文件指针的偏移) 作为定位这条消息的唯一 ID。
- ❑ replicas, 备份。kafka 可以设定每个分区的备份数量。
- ❑ Broker, 代理。是一个相对的概念, 即当只有一个 kafka 的 Server 时, 并没有代理的说法, 但当有多个 kafka 服务形成集群 (cluster) 时, 每个具体的 kafka 身份就变成了一个 Broker。
- ❑ zookeeper, “动物园管理员”。kafka 自身实现高吞吐量的核心目标, 而 zookeeper 则辅助 kafka 来管理各个消费者和在各个 broker 之间共享信息, 比如, 保存每个消费者对分区消息消费的偏移量信息。
- ❑ Leader/Follower, 领导者 / 跟随者。也是一个相对的概念, 领导者主抓分区的读取和写入, Follower 负责跟随。如有两个 broker, 1 和 2; 有 5 个分区 A、B、C、D、E。broker1 负责 A、B、C 分区的读取和写入, 则对于 A、B、C 分区来说, broker1 是领导者, broker2 是追随者。broker2 负责 D、E 分区的读取和写入, 则对于 D、E 分区来说, broker2 是领导者, broker1 是追随者。gossip 服务中的 leader 模块也是同样的做法。

1. sarama

Sarama is an MIT-licensed Go client library for Apache Kafka, 这是 sarama 库 README 的第一句话, 意思是说, sarama 是一个 kafka 的 go 客户端库。sarama 使用的基本流程如下 (产生的对象均是客户端对象):

```
//1. 创建一个kafka的综合配置对象, 该对象可以配置生产者、消费者、网络等。
brokerConfig := sarama.NewConfig()
```


//2.对配置对象逐一赋值，细节不叙，这一点是比较杂的一点。

```
brokerConfig.Producer.XXX = ...
brokerConfig.Consumer.XXX = ...
brokerConfig.Net.XXX = ...
brokerConfig.Metadata.XXX = ...
...
```

//3.根据配置创建生产者对象，其中brokers为字符串数组类型的broker的地址列表，

//在生成具体的生产者对象时，该对象已经根据brokers连接了kafka服务端。

//sarama的生产者对象有SyncProducer、AsyncProducer两种，即同步、异步两种。

//同步表现在出版一条kafka消息后会一直阻塞到收到出版成功的回复，异步则不存在阻塞。

//同步Producer

```
producer := sarama.NewSyncProducer(brokers, brokerConfig)
```

//异步Producer

```
producer := sarama.NewAsyncProducer(brokers, brokerConfig)
```

//4.根据配置创建消费者和分区消费者，参数与生产者一致。

//消费者是负责生成和管理（多个）分区消费者的对象，分区消费者是真正订阅消费消息的对象。

```
consumer := sarama.NewConsumer(brokers, brokerConfig)
```

//参数依次是哪个主题、哪个分区、从哪个offset开始（消费）

```
partitionConsumer := consumer.ConsumePartition(topic, partition, startFrom)
```

//5.生产者出版kafka消息，分区消费者订阅消费kafka消息

```
producer.SendMessage(&sarama.ProducerMessage{...})
```

```
msg_chan := partitionConsumer.Messages()
```

//从分区消费者chan中获取消息并使用

```
msg := <-msg_chan
```

...

//6.生产者，消费者关闭，为防止leak，必须进行此步操作。

```
producer.Close()
```

```
consumer.Close()
```

```
partitionConsumer.Close()
```

2. Fabric 中的 kafka

Fabric 中使用 kafka 的基本目的是在多个 peer 节点同时向 orderer 服务发送消息时，能通过 kafka 形成一个串行的消息队列（队列中消息序号唯一），然后供 cutter 进行封装成一批批消息（即 block），再将 block 写入 orderer 的 ledger 中，也就形成了块链，最后供区域内各个 peer 节点中的 leader 获取 orderer 的 ledger 中的数据，然后由 leader 将数据所在的区域中的各个 peer 节点间共享，形成了区块链。

Fabric 中关于 sarama 用到的关于 kafka 的配置集中在 orderer.yaml 中的 kafka 配置区域，configtx.yaml 中的 orderer 部分中的 kafka 部分。

sarama 中主要的对象，生产者 / 消费者都封装在 orderer/kafka/chain.go 的 chainImpl 对象中，而综合配置则封装在 orderer/kafka/consenter.go 文件的 consenterImpl 对象中。

Fabric 中的 kafka，具体来说是 orderer 中的 kafka，是一个客户端，由 sarama 实现，即通过网络连接的 kafka 服务端容器。sarama 被包装进了 orderer/kafka/chain.go 中的

chainImpl 中使用。在 chainImpl 成员中,producer 即为生产者,parentConsumer 为消费者(负责生成和管理分区消费者的),channelConsumer 为分区消费者(实际消费消息的)。

5.3 orderer 在 Fabric 中的交互流程

5.3.1 建立连接

grpc 分为客户端和服务端,对于 peer 连接 orderer 来说,peer 是客户端,orderer 为服务端。

Broadcast 服务主要集中在 peer 节点的命令中,如 peer chaincode、peer channel 命令。在 peer/common/common.go 中,GetBroadcastClientFnc 的值为 peer/common/ordererclient.go 中的 GetBroadcastClient 函数,peer chaincode 和 peer channel 命令中使用到的命令工厂中的 broadcast 客户端 BroadcastClient (在 peer/common/common.go 中的 ChaincodeCmdFactory 中)均由此函数生成。同时,peer chaincode 和 peer channel 命令执行时,都有一个 Flag-o,来指定所要连接的 orderer 服务节点的地址。当命令(第一次)执行时,peer 节点会建立与 orderer 节点的 Broadcast 服务 grpc 连接。

Deliver 服务较为复杂,对象之间相互嵌套,主要集中在 peer 节点的 gossip 服务中,且包裹在 core/deliverservice/blocksprovider/blocksprovider.go 文件中的 createClient,client 的类型是 core/deliverservice/client.go 中的 broadcastClient,createClient 不是 Deliver 服务本身,而是用于生成 Deliver 服务实例,而 core/deliverservice/deliverclient.go 中的 deliverServiceImpl 按 chainID 存储不同的 blocksProviderImpl (即每个 chain 都会有一个 client),是 gossip 服务直接使用的 deliver 服务的对象。

1) 在 peer/node/start.go 的 serve 函数中,service.InitGossipService(...) 初始化了 gossip 服务,将一个专门生成 deliver 服务对象的 deliver 工厂赋值给 gossip 服务的成员 deliveryFactory,这个 deliver 工厂的值是 gossip/service/gossip_service.go 中的 deliveryFactoryImpl。serve 中随后的 peer.Initialize(...) (core/peer/peer.go) 调用了 createChain,在 createChain 中调用了 service.GetGossipService().InitializeChannel(...) (gossip/service/gossip_service.go),传给 InitializeChannel 的最后一个参数 ordererAddresses 就是 orderer 的地址,对 gossip 服务进行了进一步初始化。

2) 在 InitializeChannel 中,使用了 1) 中 deliveryFactoryImpl 的 Service 函数,进而使用 deliverclient.NewDeliverService(config) (core/deliverservice/deliver client.go 中实现) 生成了一个 Deliver 服务实例 deliverServiceImpl,将其赋值给 gossip 服务中的成员 deliveryService,所给配置 config 中的 Endpoints、ConnFactory、ABCFactory 被分别赋值为传进来的 orderer 的地址,core/deliverservice.go 中的 DefaultConnection Factory,DefaultABCFactory。

3) 在 `InitializeChannel` 中, 无论是静态指定 leader 还是动态选举 leader, 最终都会调用 `deliveryService.StartDeliverForChannel(...)`, 即在 `core/deliverservice/deliverclient.go` 中, `StartDeliverForChannel` 会使用 `newClient->NewBroadcastClient (core/deliverservice/client.go)` 创建一个 client, 再使用 `blocksprovider.NewBlocksProvider(...)` 创建一个包含 client 的 `blocksProviderImpl`, 随后 `go d.blockProviders[chainID].DeliverBlocks()` 启动 client 的接收线程。

4) 在 `NewBroadcastClient` 中, `createClient` 就被赋值为上述 config 的 `ABCFactory`, 而生成的 `broadcastClient` 是 `blocksProviderImpl` 中的成员 `client`。即 `blocksProviderImpl` 中的 `createClient` 的值是 `core/deliverservice/deliverclient.go` 中的 `DefaultABCFactory`, `DefaultABCFactory` 使用了 `protos/orderer/ab.pb.go` 中生成的默认的 `AtomicBroadcastClient` (包含默认的 `Deliver` 客户端)。启动 client 的接收线程后, 在 client 接收消息时, `core/deliverservice/client.go` 中的 `Recv()` 会执行 `try(...)`, 进而执行 `bc.doAction(action)`, 在 `doAction` 中会查看 client 是否已经和 orderer 通过 grpc 连接上, 若未连接则会执行 `connect()` 进行 grpc 连接。

5) 当 peer 节点的容器启动的时候, peer 节点的 gossip 服务随着 peer node start 命令启动, 其中角色是 leader 的 peer 节点的 gossip 服务中使用到的 deliver, 服务对象 `deliverServiceImpl` 也会使用默认生成的 `Deliver` 客户端, 通过 grpc 连接 orderer 节点并启动循环接收的线程。

5.3.2 Broadcast

以 peer chaincode instantiate 为例, 假设名为 peerA 的节点将 example02 部署完毕后, 生成了一个 `Envelope` 发送给 orderer。cf 即为 peer 的命令工厂 `ChaincodeCmdFactory` 的实例, cf 成员 `BroadcastClient` 即为使用 `/protos/orderer/ab.pb.go` 中默认生成的 `Broadcast` 服务客户端 `AtomicBroadcast_BroadcastClient`。cf.`BroadcastClient.Send(env)`, 通过 grpc 将消息以 `Envelope` 的形式发送到 orderer 节点中。

5.3.3 orderer

1) `orderer/server.go` 中的 `Broadcast` 收到 peer 点发来的 `Envelope` 消息, 直接交由成员 `bh` 的 `Handle` 处理。

2) `bh` 原型为 `orderer/common/broadcast/broadcast.go` 中的 `handlerImpl`, `Handler` 函数也在这里实现。在 `Handler` 函数中, 会在 for 中循环接收来自 peer 节点的消息: (a) `msg, err := srv.Recv()` 接收消息。(b) `Unmarshal` 并检查 `Envelope` 消息中的一些字段 (如 `ChannelId`), 如果是 `HeaderType_CONFIG_UPDATE` 类型的消息, 则会将消息经过 `bh.sm.Process(msg)`, 实际是调用 `orderer/configupdate/configupdate.go` 中的 `Process` 对消息进行进一步加工, 将消息加工成一个 orderer 可以处理的交易消

息(加工成创建新 chain 的配置或对已存在的 chain 更新配置)。(c) `support, ok := bh.sm.GetChain(chdr.ChannelId)`, 获取消息对应 `ChannelId` 的链的 `chainSupport`, 然后执行函数 `support.Filters().Apply(msg)`, 使用该 `chainSupport` 的过滤器过滤该消息, 以查看该消息是否达标, 这里没有使用过滤器一并返回 `committer` 集合, 这是第一次过滤。(d) `support.Enqueue(msg)`, 进而调用了 `support` 对应的 chain 的 `cs.chain.Enqueue(env)` 处理消息, 这个 chain 是对应的 `consenter` (这里以 `kafka` 为例进行介绍)的 `HandleChain` 生成的(在创建 `chainSupport` 的时候, 在 `orderer/multichain/chainsupport.go` 的 `newChainSupport` 中, 执行 `cs.chain, err = consenter.HandleChain(cs, metadata)`)。

3) 消息进入 `orderer/kafka/chain.go` 的 `Enqueue(env)`, 在最外层的 `select-case` 中, 如果 `startChan` 已经被 `close`, 则说明 `orderer` 的 `kafka` 客户端部分(即 `orderer/kafka/chain.go` 的 `chainImpl`)已经一切准备就绪, 则会进入 `case <-chain.startChan`: 分支, 否则进入 `default`: 分支, 不处理消息而返回。在 `case <-chain.startChan`: 分支中, 如果该 chain 没有停止, 即 `haltChan` 没有被 `close`, 则会进入 `default`: 分支, 消息也会在此进行处理。

4) 在这个 `default`: 分支中: (a) `message := newProducerMessage(chain.channel, payload)` 把消息打包成 `kafka` 消息类型, 也是使用 `sarama` 预定义的 `Producer Message` 类型, `Topic` 定义了消息的主题——`Key` 和 `Value`, `key` 为分区号, `value` 为 `Marshal` 过的 `peer` 点发来的原始的消息。(b) `chain.producer.SendMessage(message)`, 使用 `sarama` 的生产者对象将 `kafka` 消息发送到 `kafka` 服务端。

5) `kafka` 服务端(即 `kafka` 容器)这个“暗盒”接收到消息并生产消息。

6) 在之前 chain 启动时接收的 `kafka` 服务端消息的线程中, 即 `orderer/kafka/chain.go` 中的 `processMessagesToBlocks()`, `kafka` 服务端生产的消息从 `case in, ok := <-chain.channelConsumer.Messages()` 处被消费出来, 得到 `ConsumerMessage` 类型的消费消息, 并进入该分支。

7) 在 `case in, ok := <-chain.channelConsumer.Messages()` 分支中, 首先出现的一个 `select-case` 是一个关于 `errorChan` 通道的开关, `errorChan` 通道在 chain 初始化之初是关闭的, 而后在 `startThread()` 的 `chain.errorChan = make(chan struct{})` 中再度生成, 算是再度开启。如果 `errorChan` 是关闭的, 则在 `select-case` 中会因进入 `case <-chain.errorChan`: 分支而返回, 否则进入 `default`: 分支, 程序继续。随后进入一个 `switch-case`, 根据解压消费消息所携带的数据的类型, 进入不同分支来处理消息。正常来说, 数据会进入 `case ab.KafkaMessage_Regular`: 分支, 然后被 `processRegular(...)` 处理。这里注意两个数据: (a) 一是传入 `processRegular(...)` 的消费消息的 `in.Offset`, 该值是消息在 `kafka` 服务端分区中的 `offset`, 在整个生产消费过程中序列化排序会依次分配给每个消息, 因此每个消息都具有唯一的 `offset`, 也因此 `in.Offset` 可以代表每个具体的消费消息。在 `block` 的结构中, 一方面, `Metadata` 是一个数组, 用来存储

与 block 块有关的基础信息, 其中下标 `BlockMetadataIndex_ORDERER` 处存储的就是 block 中最后一条消息的 `offset` 值+1 (也相当于下一个 block 中第一条消息的 `offset`); 另一方面, block 中实际存储数据的 `Data` 是一个 `[]byte` 数组, 序列化过后的消息都被二进制化后依次存放在这个数组中, 我们可以算出 block 中具体有多少条消息。因此通过 `< offset+1` 的条件, 就可以准确定位 block 中的每条消息。(b) 二是传入 `processRegular(...)` 的 `time` 计时器。

8) 在 `processRegular()` 中, 将原始数据 (即 `peer` 节点发来的消息) 从 `kafka` 类型消息中分解出来后, 交由 `blockcutter` 模块的 `ordered` 进行分块处理。

9) 消息进入 `orderer/common/blockcutter/blockcutter.go` 的 `Ordered(env)`, 先经由 `r.filters.Apply(msg)` 进行了第二次的过滤, 然后就是一个根据配置信息进行分割成 block 的过程。这里假设此时满足生成一批消息的条件, 该批消息和对应的 `committer` 集合一同被返回。

10) 重回 `processRegular()`, 当返回了一或两批消息, 程序会进入 `for i, batch := range batches` 循环依次处理每批消息。在循环中: (a) 计算当批消息的 `offset` 值 (即最后一条消息的 `offset` 值+1), `offset := receivedOffset - int64(len(batches)-i-1)`, (b) `block := support.CreateNextBlock(batch)`, 将当批消息打包成 block, 至此消息正式成块。(c) `support.WriteBlock(block, committers[i], encodedLastOffsetPersisted)`, 将 block 对应的 `committer` 集合, 然后将 block 写入账本。

11) 这里单独描述一下传入 `processRegular()` 中的 `time` 计时器和该计时器控制的主动 `Cut` 操作。调用 `processRegular(...)` 是 `chainImpl` 启动的 `processMessagesToBlocks`, 而消息来源于 `kafka` 服务器, 如果 `kafka` 服务器长时间没有消息被消费出来, 可能是由于 `kafka` 服务器出了故障, 也可能是因为客户端没有新的生产消息发给 `kafka` 服务器。在这种情况下, `blockcutter` 可能已经缓存了一些之前的消息, 为了不使这部分消息丢失并将其及时记录到账本中 (比如 `kafka` 处理的数据量很小或消息流很不稳定, 一条消息后很长时间都不来下一条消息, 会造成已缓存的数据不能及时记录到账本, 有丢失的风险。也有交易已经产生了好长时间, 但是包裹交易的消息一直悬空在 `blockcutter` 缓存中无法被消费, 进而无法被记录和查询), `configtx.yaml` 中配置的 `BatchTimeout` 规定了超时时间, 默认 2s。当下一条消息超过 2s 还没被消费出来, 将会主动 `Cut` 操作将现有缓存打包成一个 block 写入账本。具体的操作是: (a) `timer` 初始状态为 `nil`, 当消息进入 `processRegular()` 中, 若缓存进入 `blockcutter`, 则会进入 `if ok && len(batches) == 0 && *timer == nil` 分支, 设 `timer` 计时器为 2s 后触发然后返回。(b) 下一条消息若在 2s 之前再次进入 `processRegular()`, 若仍是被放入缓存, 则依然会进入 `if ok && len(batches) == 0 && *timer == nil` 重置 `timer` 为 2s 计时然后返回。(c) 当 `timer` 没有及时被重置, 即超过了 2s, `processMessagesToBlocks` 会进入 `case <-timer:` 分支, 调用 `sendTimeToCut` 向 `kafka` 服务器发送一条 `TimeToCutMessage` 消息 (包裹着 `chain.lastCutBlockNumber+1`, 即若主

动 Cut, 会使用 block 的序号), 接着 kafka 服务器收到, 然后被消费出来 (这里存在一个时间过程), 这条 TimeToCutMessage 消息会再次进入 processMessagesToBlocks 的 case in, ok := <-chain.channelConsumer.Messages(): 分支, 进而进入 case ab.Kafka Message_TimeToCut: 分支, 调用 processTimeToCut(...) 处理这条消息, 也就是主动 Cut。(d) 在 processTimeToCut(...) 中, 如果此时 if ttcNumber == *lastCut BlockNumber+1 (ttcNumber 即为 (c) 点 TimeToCutMessage 消息包裹的 block 序列号), 说明在 c 点提到的时间过程中, chain.lastCutBlockNumber 的值没变, 也就是没有发生新的 Cut, 因为一旦有新的 Cut 发生, chain.lastCutBlockNumber 就会增 1。没有发生新的 Cut, 则说明 blockcutter 中现在缓存的数据依然是发送 TimeToCutMessage 消息时的数据, 或者有新数据但是依旧没有达到 Cut 的条件。此时就会进行主动的 Cut, blockcutter 中现有的缓存被打包出来后清空, timer 计时器也会被重置为 nil。另外, 在 processRegular(...) 中, 当常规触发了 Cut 后, blockcutter 中将不存在缓存消息, 之后会进入 if len(batches) > 0 分支, 会将 timer 计时器置为 nil, 在 processMessagesToBlocks 中的 for-select-case 等待再次从 kafka 中消费出消息。

12) 在 block := support.CreateNextBlock(batch) 和 support.WriteBlock(block, committers[i], encodedLastOffsetPersisted) 中, 调用了 support 中的 ledger 创建 block 和写入 block, 在写入 block 之前会逐一执行该 block 对应的 committer 集合。至此, orderer 端处理 peer 节点发送来的消息的流程结束。

5.3.4 Deliver

Deliver 服务较 Broadcast 服务复杂, 我们知道最终 orderer 中 ledger 账本的数据会通过 Deliver 服务推送 leader 节点。

1) Deliver 服务是建立在 grpc 之上的, 且 orderer 为 Deliver 的服务端, 则可以推断是 peer 节点向 orderer 索要 block 数据。gossip 服务的索要行为的对象是 core/deliverservice/requester.go 中的 blocksRequester, 索要行为发生在 RequestBlocks(...) 函数中, 该函数传入一个包裹了高度值的 LedgerInfo, 若该高度值为向 orderer 索要的开始的 block 的序列号, 且高度值 > 0, 则调用 seekLatestFromCommitter, 否则调用 seekOldest()。两个函数过程基本一致, 先创建一个包装了 SeekInfo 信息 (即索要的 block 的起止范围信息) 的 HeaderType_CONFIG_UPDATE 类型的 Envelope 消息, 然后 b.client.Send(env) 向 orderer 端的 Deliver 服务端索要 block。注意, 这两个函数索要的 block 范围的起点可能不一样, 但是止点都一致为 math.MaxUint64, 即最大极限值, 这相当于向 orderer 端索要现在以及将来所产生的所有 block。

2) 索要全部的 block 的行为在何时发生? (a) 在 core/deliverservice/deliverclient.go 的 newClient 中, broadcastSetup := func(...) 调用了 RequestBlocks(...)。(b) broadcastSetup 作为一个参数通过 NewBroadcastClient 赋值给

了 bClient (原型是 core/deliverservice/client.go 中的 broadcastClient) 的成员 onConnect。(c) 随后 bClient 通过 newClient 返回, 在 StartDeliverForChannel(...) 中的 blocksprovider.NewBlocksProvider(...), bClient 赋值给了对应 chainID 的 BlocksProvider (core/deliverservice/blocksprovider/blocksprovider.go) 的成员 client。(d) 接着 go d.blockProviders[chainID].DeliverBlocks(), 调用了 b.client.Recv(), 这个 client, 就是 c 点提到的 bClient。(e) Recv() 在 core/deliverservice/client.go 中, 调用了 bc.try(...), 传给 try(...) 参数是一个执行接收动作的函数, 即使用 bc.BlocksDeliverer.Recv() 接收消息 (此时 BlocksDeliverer 还没有被赋值)。(f) 在 try(...) 中, for 循环不断尝试执行 bc.doAction(action), 这个 action 是上一步传入的接收消息的动作。在 doAction(action) 中, 会首先查看连接 conn 是否已经建立, 若没有建立, 则会调用 bc.connect() 建立连接。随后 resp, err := action() 执行 action 动作。(g) 建立连接章节 Deliver 部分, 追溯一下可知, 在 connect() 中, 使用了 conn, endpoint, err := bc.prod.NewConnection() 建立了一个连接 orderer 节点的 grpc 连接, abc, err := bc.createClient(conn).Deliver(ctx) 使用建立的 grpc 连接创建了一个 Deliver 服务客户端, 随后调用了 bc.afterConnect(conn, abc, cf)。(h) 在 afterConnect 中, 将创建与 orderer 节点的连接和 Deliver 客户端, 分别赋值给 broadcastClient 的 conn (可使之后的所有 doAction(...) 中的 connect() 不再执行) 和 BlocksDeliverer (在 e 时 BlocksDeliverer 还没有被赋值), 然后调用 bc.onConnect(bc), 这个 onConnect 就是 a、b 中的 broadcastSetup, 即在这里发生了索要行为, 向 orderer 节点索要所有的 block。(i) 随后开始返回, 重新返回到 doAction 中, 与 orderer 节点的连接建立并发送了索要请求后, 继续开始执行动作 resp, err := action(), 这个 action 即为开始接收 orderer 节点发来的 block 数据 (e)。(j) 以上步骤都是随着 gossip 服务运行起来, 调用 deliverServiceImpl 对象的 StartDeliverForChannel 在启动的 go d.blockProviders[chainID].DeliverBlocks() 中的 for 循环是持续调用的, 其中建立连接和索要行为只会发生一次。

3) 在 2) 的 h 点中执行 bc.onConnect(bc) 后, 即执行 core/deliverservice/requester.go 中的 RequestBlocks 后, orderer 节点在 orderer/common/deliver/deliver.go 的 Handle 中的 envelope, err := srv.Recv() 处被接收, 之后: (a) 进行一系列从 Envelope 的解压收取数据的操作, 然后 chain, ok := ds.sm.GetChain(chdr.ChannelId) 获取 Envelope 中对应 chainID 的 chainSupport。(b) erroredChan := chain.Errorred(), select {case <- erroredChan: ..., 这也是一个 erroredChan 控制的开关, 此刻 erroredChan 开关处于开启状态, 程序将继续向下走。(c) sigfilter.New(...), sf.Apply(envelope), 创建了一个签名过滤对象验证 Envelope 消息。(d) 从 Envelope 解压出来携带的索要的 block 的起止信息 SeekInfo, cursor, number := chain.Reader().Iterator(seekInfo.Start), 根据起止信息创建一个账本的查询迭代器

cursor, 这个迭代器是在 `orderer/ledger/file/impl.go` 中定义的 `fileLedgerIterator`, 存储了起点, 有一个特性是 `Next()` 会在迭代到还没有写入到账本 block (即当前账本最新的 block 的下一个 block) 时阻塞等待, 一直等待到该 block 被写入到账本中。(e) 在 for 循环中, `block, status := cursor.Next()`, 不断使用迭代器 `cursor` 获取一个 block, 随后 `sendBlockReply(srv, block)` 向 Deliver 客户端回复该 block。(f) 向 Deliver 客户端回复一条 block 后, 会进行 `if stopNum == block.Header.Number` 判断, 若刚刚发送的 block 已经是客户端索要的最后一块 block, 则 `break` 跳出循环等待客户端下次索要行为。但是前面已经提到, gossip 服务索要的 block 的止点是 `math.MaxUint64`, 即 `stopNum` 的值是 `math.MaxUint64`, 所以向 gossip 服务发送 block 的 for 循环“永远”不会退出。

4) orderer 的 Deliver 服务端向 peer 节点的 Deliver 客户端发送 block, block 进入 2) 的 i 点的 `doAction` 中, `resp, err := action()` 接收到 block, 返回至 `try(...)`, 再返回至 `Recv()`, 然后返回 `core/deliverservice/blocksprovider/blocksprovider.go` 的 `DeliverBlocks()` 中的 `msg, err := b.client.Recv()`, 最后进入 `case *orderer.DeliverResponse_Block`: 分支: (a) `gossipMsg := createGossipMsg(b.chainID, payload)` 将 block 包装成 gossip 服务能处理的消息类型。(b) `b.gossip.AddPayload(b.chainID, payload)` 将 block 添加到 peer 节点的 gossip 服务模块的本地, 目的在于添加块到本地的 ledger 中。再次强调, 接收 block 的是 peer 节点中的 leader, 所以这里的 gossip 服务模块是 leader 的。(c) `b.gossip.Gossip(gossipMsg)`, leader 的 gossip 服务模块向其他 peer 节点散播 block。

5) 向 orderer 的 Deliver 服务端索要 block 消息, 除了 gossip 服务, 还有 peer channel 的几个子命令, 但 peer channel 索要的都是一定量的 block, 即非持续性的, 只是 Deliver 服务端最后会进入 `if stopNum == block.Header.Number` 分支跳出回复 block 的 for 循环 e 点的描述)。peer 节点中的 leader 的 gossip 服务启动之后就建立了与 orderer 的 Deliver 服务的连接 (这期间 peer 节点只会在开始的时候向 orderer 的 Deliver 服务索要一次 block), 之后当 orderer 中的账本中存在 block 数据后, 就开始主动向 leader 的 Deliver 客户端发送 block 数据, 这个推送行为将一直持续。leader 的 Deliver 客户端收到 block 流之后会使用 gossip 服务向自身和其他 peer 节点散播这些 block 数据。另外, peer channel 等命令也会向 Deliver 服务端请求某一段 block 数据, 但该推送是非持续性的。

chaincode 的设计与实现

chaincode 是 Fabric 实现智能合约的方式，利用容器技术将智能合约放置在容器中运行，进行调用。

6.1 chaincode 生命周期管理

Hyperledger Fabric API 允许用户与区块链网络中的各种节点（peer 节点、orderer 节点、MSP 节点）进行交互，同时还可以在背书节点上安装、实例化及升级 chaincode。Hyperledger Fabric 特定语言的 SDK 工具将 Hyperledger Fabric API 的细节抽象了出来，便利了应用的开发过程；当然它也能用于管理 chaincode 生命周期。除此之外，Hyperledger Fabric API 可以直接由 CLI（命令行）访问。

我们提供了四个管理 chaincode 生命周期的命令：package, install, instantiate, upgrade。在未来的版本中，我们考虑添加 stop 和 start 交易的指令，以便方便地停止与重启 chaincode，而不用非要真正卸载它才行。在成功安装与实例化 chaincode 后，chaincode 就处于运行状态，接着就可以用 invoke 交易指令来处理交易了。一段 chaincode 可以在安装后的任何时间被更新。

6.1.1 打包

chaincode 包具体包含以下三个部分：

- chaincode 本身由 ChaincodeDeploymentSpec 或 CDS 定义。CDS 根据代码以及一些其他属性（名称，版本等）来定义 chaincode。
- 一个可选的实例化策略，该策略可以被背书策略描述。

- ❑ 一组表示 chaincode 所有权的签名。channel 上 chaincode 实例化交易的创建者可被 chaincode 的实例化策略验证。

1. 创建包

打包 chaincode 有两种方式。当 chaincode 有多个所有者的时候,此时需要让 chaincode 包被多个所有者签名。这种情况下需要创建一个被签名的 chaincode 包 (SignedCDS), 这个包依次被每个所有者签名。

另一种,只有一个节点的签名(该节点执行 install 交易)。

第一种情况,要创建一个签名过的 chaincode 包,请使用下面的指令:

```
peer chaincode package -n mycc -p github.com/hyperledger/fabric/
examples/chaincode/go/chaincode_example02 -v 0 -s -S -i "AND('OrgA.
admin')" ccpack.out
```

-s 选项创建了一个可被多个所有者签名的包,而非简单地创建一个 CDS。如果使用 -s,那么当其他所有者要签名时,-S 也必须同时使用。否则,该过程将创建一个仅包含实例化策略的签名 chaincode 包 (SignedCDS)。

-s 选项可以使在 core.yaml 文件中被 localMspid 相关属性值定义好的 MSP 对包进行签名。

-s 选项是可选的。但是如果创建了一个没有签名的包,那么它就不能被任何其他所有者用 signpackage 指令进行签名。

-i 选项也是可选的,用来为 chaincode 指定实例化策略。实例化策略与背书策略格式相同,它指明谁可以实例化 chaincode。在上面的例子中,只有 OrgA 的管理员才有资格实例化 chaincode。如果没有提供任何策略,那么系统会采用默认策略,该策略只允许 peer 节点 MSP 的管理员去实例化 chaincode。

2. 包的签名

一个在创建时就被签名的 chaincode 包可以交给其他所有者进行检查与签名。

ChaincodeDeploymentSpec 可以选择被全部所有者签名并创建一个 SignedChaincode DeploymentSpec (SignedCDS), SignedCDS 包含三个部分:

- ❑ CDS 包含 chaincode 的源码、名称与版本。
- ❑ 一个 chaincode 实例化策略,其表示为背书策略。
- ❑ chaincode 所有者的列表,由 Endorsement 定义。



注意 当 Chaincode 在某些 channel 上实例化时,背书策略在外部定义,并提供相应的 MSP。如果没有明确实例化策略,那么默认的策略是 channel 的任意管理员(执行实例化)。

每个 (Chaincode 的) 所有者通过将 ChaincodeDeploymentSpec 与其本人的身份信息 (证书) 结合并对组合结果签名来认证 ChaincodeDeploymentSpec。


一个 Chaincode 所有者可以对一个之前创建好的带签名的包进行签名，具体使用如下指令：

```
peer chaincode signpackage ccpack.out signedccpack.out
```


指令中的 `ccpack.out` 和 `signedccpack.out` 分别是输入与输出包。`signedccpack.out` 则包含一个用本地 MSP 对包进行的附加签名。

6.1.2 安装 chaincode

`install` 交易的过程会将 chaincode 的源码以一种被称为 `ChaincodeDeploymentSpec` (CDS) 的规定格式打包，并把它安装在一个将要运行该 chaincode 的 peer 节点上。

 **注意** 请务必在一条 channel 上每一个要运行你 chaincode 的背书节点上安装你的 chaincode。


如果只是简单地给 `install` API 一个 `ChaincodeDeploymentSpec`，它将使用默认实例化策略并添加一个空的所有者列表。

 **注意** chaincode 应该只被安装在 chaincode 所有者的背书节点上，便于 chaincode 逻辑对整个网络的其他成员保密。其他没有 chaincode 的成员将无权成为 chaincode 影响下的交易的认证节点 (endorser)。也就是说，他们不能执行 chaincode。不过，他们仍可以验证交易并提交到账本上。

安装 chaincode 时，系统会发送一条 Signed Proposal 到生命周期系统 chaincode (LSCC)。例如，使用 CLI 安装简单的账本管理 chaincode 中的 `sacc` chaincode 样例时，命令如下：

```
peer chaincode install -n asset_mgmt -v 1.0 -p sacc
```

在 CLI 内部会为 `sacc` 创建 `SignedChaincodeDeploymentSpec`，并将其发送到本地 peer 节点。这些节点会调用 LSCC 上的 `Install` 方法。上述的 `-p` 选项指明 chaincode 的路径——必须在用户的 `GOPATH` 目录下（比如 `$GOPATH/src/sacc`）。完整的命令选项详见 CLI 部分。

 **注意** 为了在 peer 节点上安装 (chaincode)，Signed Proposal 的签名必须是来自 peer 节点本地 MSP 的管理员中的一位。

6.1.3 实例化 chaincode

实例化交易通过调用生命周期系统 chaincode (LSCC) 在一个 channel 上创建并初始化一段 chaincode。下面是一个 chaincode-channel 绑定的具体过程：一段 chaincode 可能会与任意数量的 channel 绑定并在每个 channel 上独立运行。chaincode 在多个 channel 上相互独立，对于每个提交交易的 channel，其状态都是互不影响的。

一个实例化交易的创建者必须符合在 SignedCDS 中 chaincode 的实例化策略，且必须充当 channel 的写入器（为 channel 创建配置的一部分）。这对于 channel 的安全至关重要，这样可以防止恶意实体在未绑定的 channel 上部署 chaincode，也能防止间谍成员在未绑定的 channel 上执行 chaincode。

例如，我们提到过默认的实例化策略是任何 channel MSP 的管理员（可以执行），所以 chaincode 创建者要实现实例化交易，其本身必须是 channel 管理员的一员。当交易提议到达背书成员时，它会验证创建者的签名是否符合实例化策略。在交易被提交到账本之前的交易验证阶段，会重复上述操作。

实例化交易的过程还会为 channel 上的 chaincode 建立背书策略。背书策略描述了交易的相关认证要求，使交易能被 channel 中的成员认可。

例如，使用 CLI 去实例化 sacc chaincode 并初始化 john 的状态为 0，具体指令如下：

```
peer chaincode instantiate -n sacc -v 1.0 -c '{"Args":["john","0"]}' -P "OR
('Org1.member','Org2.member')"
```



注意 上述背书策略（CLI 使用波兰表示法）向 Org1 或 Org2 的成员询问所有 sacc 处理的交易。为确保交易有效，Org1 或 Org2 必须为调用 sacc 的结果签名。

在成功实例化后，channel 上的 chaincode 就进入激活状态，并时刻准备执行任何 ENDORSER_TRANSACTION 类型的交易提议。交易会在到达背书节点的同时被处理。

6.1.4 升级 chaincode

一段 chaincode 可以通过更改版本（SignedCDS 的一部分）来随时更新。至于 SignedCDS 的其他部分，如所有者及实例化策略，都是可选的。但是 chaincode 的名称必须一致，否则会成为两个完全不同的 chaincode。

在升级之前，chaincode 的新版本必须安装在需要它的背书节点上。升级是一个类似于实例化交易的交易，它会将新版本的 chaincode 与 channel 绑定。其他与旧版本绑定的 channel 则仍旧运行旧版本的 chaincode。升级交易只会一次影响一个提交它的 channel。




注意 由于多个版本的 chaincode 可能同时运行，所以升级过程不会自动移除旧版本，用户必须亲自处理。

升级交易与实例化交易有一处微妙的区别：升级交易采用当前的 chaincode 实例化策略进行检查，而非比对新的策略（如果指定了的话）。这是为了确保只有当前实例化策略指定的已有成员才能升级 chaincode。

在升级过程中，chaincode 的 Init 函数会被调用以执行数据相关的操作，或者重新初始化数据，所以要多加小心避免在升级 chaincode 时重设状态信息。

6.1.5 停止与启动

 **注意** 停止与启动生命周期交易的功能还没实现，不过你可以通过移除 chaincode 容器以及从每个背书节点删除 SignedCDS 包来停止 chaincode。具体而言，就是删除所有主机或虚拟机上 peer 节点运行于其中的 chaincode 的容器，随后从每个背书节点删除 SignedCDS。


为了从 peer 节点删除 CDS，你应该需要先进入 peer 节点的容器内。下面提供一个可以执行此功能的脚本：

```
docker rm -f <container id>
rm /var/hyperledger/production/chaincodes/<ccname>:<ccversion>
```

停止功能在以受控的方式进行升级的流程中非常有用，特别是在进行升级前，一段 channel 上所有节点的 chaincode 都可被停止。

6.1.6 CLI

利用命令行可以理解 Fabric 中 chaincode 的运行原理。

 **注意** 下面是可用的 CLI 指令，请在一个运行 fabric-peer 的 Docker 容器中执行以下指令：

```
docker run -it hyperledger/fabric-peer bash
# peer chaincode --help
```

输出如下：

Usage:

peer chaincode [command]

Available Commands:

```
install      Package the specified chaincode into a deployment spec and save it
              on the peer's path.
instantiate  Deploy the specified chaincode to the network.
invoke       Invoke the specified chaincode.
package      Package the specified chaincode into a deployment spec.
query        Query using the specified chaincode.
signpackage  Sign the specified chaincode package
upgrade      Upgrade chaincode.
```

Flags:

```
--cafile string  Path to file containing PEM-encoded trusted certificate(s)
                  for the ordering endpoint
-C, --chainID string  The chain on which this command should be executed (default
                      "testchainid")
-c, --ctor string    Constructor message for the chaincode in JSON format (default
                      "{}")
```



```

-E, --escv string      The name of the endorsement system chaincode to be used
                        for this chaincode
-l, --lang string      Language the chaincode is written in (default "golang")
-n, --name string      Name of the chaincode
-o, --orderer string    Ordering service endpoint
-p, --path string      Path to chaincode
-P, --policy string     The endorsement policy associated to this chaincode
-t, --tid string       Name of a custom ID generation algorithm (hashing and
                        decoding) e.g. sha256base64
--tls                  Use TLS when communicating with the orderer endpoint
-u, --username string   Username for chaincode operations when security is enabled
-v, --version string    Version of the chaincode specified in install/
                        instantiate/upgrade commands
-V, --vscv string      The name of the verification system chaincode to be used
                        for this chaincode

```

Global Flags:

```

--logging-level string    Default logging level and overrides, see
                        core.yaml for full syntax
--test.coverprofile string Done (default "coverage.cov")

```

Use "peer chaincode [command] --help" for more information about a command.

为方便在脚本应用程序里使用，peer 指令失败时总会返回一个非 0 值。

chaincode 的指令示例如下：

```

peer chaincode install -n mycc -v 0 -p path/to/my/chaincode/v0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a", "b", "c"]}' -C mychannel
peer chaincode install -n mycc -v 1 -p path/to/my/chaincode/v1
peer chaincode upgrade -n mycc -v 1 -c '{"Args":["d", "e", "f"]}' -C mychannel
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","e"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile $ORDERER_CA -C mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}'

```

6.2 chaincode 原理浅析

6.2.1 什么是 chaincode

chaincode 是一段由 Go 语言编写（支持其他编程语言，如 Java），并能实现预定义接口的程序。chaincode 运行在一个受保护的 Docker 容器当中，与背书节点的运行互相隔离。chaincode 可通过应用提交的交易对账本状态初始化并进行管理。

一段 chaincode 通常处理由网络中的成员一致认可的业务逻辑，通常用“智能合约”来代指 chaincode。一段 chaincode 创建的（账本）状态是与其他 chaincode 互相隔离的，故不能被其他 chaincode 直接访问。不过，如果是在相同的网络中，一段 chaincode 在获取相应许可后则可以调用其他 chaincode 来访问它的账本。

6.2.2 ChaincodeSupport 服务

ChaincodeSupport 服务为每个 peer 提供了 chaincode 操作的支持。/fabric/peer/node/start.go 文件中的 serve 函数通过 registerChaincodeSupport(peerServer.Server()) 为 peerServer 注册 ChaincodeSupport 服务。

ChaincodeSupport 的服务原型和生成的定义在 /fabric/protos/peer/ 下的 chaincode_shim.proto 和 chaincode_shim.pb.go 中, 核心的实现代码在 /fabric/core/chaincode/chaincode_support.go 中。主要定义的是一个 rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {} 服务。该服务实现客户端和服务端 ChaincodeMessage 类型流数据的交换。用于服务端流数据交换的 grpc 流服务接口为 /fabric/protos/peer/chaincode_shim.pb.go 中的 ChaincodeSupport_RegisterServer, 在 /fabric/core/container/ccintf/ccintf.go 中有对应用于容器内部间的流接口 ChaincodeStream。

ChaincodeSupport 服务以单例模式运行, 该单例对象定义在 chaincode_support.go 中的 var theChaincodeSupport *ChaincodeSupport。ChaincodeSupport 对象自身存储一系列配置值, 而接收和处理客户端 ChaincodeMessage 类型消息的任务被委托给了一个 Handler 对象。

```
//生成的收发的数据类型
type ChaincodeMessage struct {
    Type          ChaincodeMessage_Type
    Timestamp     *google_protobuf1.Timestamp
    Payload       []byte
    Txid          string
    Proposal      *SignedProposal
    ChaincodeEvent *ChaincodeEvent
}
//proto中ChaincodeSupport服务原型
service ChaincodeSupport {
    rpc Register(stream ChaincodeMessage) returns (stream ChaincodeMessage) {}
}
//生成的服务端流接口
type ChaincodeSupport_RegisterServer interface {
    Send(*ChaincodeMessage) error
    Recv() (*ChaincodeMessage, error)
    grpc.ClientStream
}
```

6.2.3 FSM

FSM 是 Finite State Machine (有限状态机) 的缩写, 是 ChaincodeSupport 服务使用到的一个第三方库。FSM 将一个事物从状态 A 向状态 B 的转化看作一个事件, 并可以设置在进入/离开某个状态时自动调用的时机函数。每个状态事件、状态、时机函数都用字符串关

键字表示。示例用法如下：

```
//创建一个状态机
//三个参数：1.默认状态 2.定义状态事件 3.定义状态转变时调用的函数
fsm := fsm.NewFSM(
    "green",
    fsm.Events{
        //状态事件的名称    该事件的起始状态Src    该事件的结束状态Dst
        //即：状态事件warn（警告事件）表示事物的状态从状态green到状态yellow
        {Name: "warn", Src: []string{"green"}, Dst: "yellow"},
        {Name: "panic", Src: []string{"yellow"}, Dst: "red"},
        {Name: "calm", Src: []string{"red"}, Dst: "yellow"},
    },
    //状态事件调用函数，在此称为时机函数。关键字用 '_' 隔开，格式是："调用时机_事件或状态"
    //before表示在该事件或状态发生之前调用该函数，如"before_warn"，表示在warn
    //这个状态事件发生前调用这个函数。"before_yellow"表示进入yellow状态之前调用
    //该函数。
    //依此类推，after表示在...之后，enter表示在进入...之时，leave表示在离开...
    //之时。
    fsm.Callbacks{
        //fsm内定义的状态事件函数，关键字指定的是XXX_event和XXX_state
        //表示任意的状态或状态事件
        "before_event": func(e *fsm.Event) {
            fmt.Println("before_event")
        },
        "leave_state": func(e *fsm.Event) {
            fmt.Println("leave_state")
        },
        //根据自定义状态或事件所定义的状态事件函数
        "before_yellow": func(e *fsm.Event) {
            fmt.Println("before_yellow")
        },
        "before_warn": func(e *fsm.Event) {
            fmt.Println("before_warn")
        },
    },
)
//打印当前状态，输出是默认状态green
fmt.Println(fsm.Current())
//触发warn状态事件，状态将会从green转变到yellow
//同时触发"before_warn"、"before_yellow"、"before_event"、"leave_state"函数
fsm.Event("warn")
//打印当前状态，输出状态是yellow
fmt.Println(fsm.Current())
```

chaincode 所能提供的函数调用有 Launch、Register、Execute、HandleChaincodeStream、stop。

6.2.4 Register

ChaincodeSupport 对象挂载的 Register 函数最终调用的是 /fabric/core/chaincode/handler.go 中的 HandleChaincodeStream 函数。在 HandleChaincodeStream 函数中通过：


```
handler := newChaincodeSupportHandler(chaincodeSupport, stream)
handler.processStream()
```

创建了一个 Handler，然后调用 Handler 的 processStream 函数对客户端发送的流数据进行了处理。newChaincodeSupportHandler 函数传入两个参数，一个是 chaincodeSupport，一个是 stream。前者是 Register 服务所在的 ChaincodeSupport 对象自身，赋值给了 Handler 对象成员 chaincodeSupport，目的是为了使 Handler 对象处理接收数据时能够使用 ChaincodeSupport 对象的服务；后者是 Register 服务的 grpc 流接口，赋值给了 Handler 对象成员 ChatStream，为了 Handler 能够从客户端接收到数据。

6.2.5 Handler

newChaincodeSupportHandler 创建并初始化了一个 Handler，初始化的成员有：ChatStream——grpc 流服务接口，是用 Register 函数传进来的；chaincodeSupport——chaincodeSupport 自身；nextState——状态通道；FSM——状态机；policyChecker——策略检查器。

6.2.6 processStream

processStream 用 recv 标识，与 | errc | msgAvil | nextState | keepalivetimer | 四个频道和 select 三者相互配合，形成了对客户端消息的接收控制。然后调用 HandleMessage、serialSend、serialSendAsync 处理接收到的消息。

- ❑ errc。错误频道。
- ❑ msgAvil。ChaincodeMessage 频道。
- ❑ nextState。包含 ChaincodeMessage 的频道。
- ❑ keepalivetime。心跳频道。

6.2.7 HandleMessage

HandleMessage 处理 ChaincodeMessage 数据的方式完全是由 Handler 中的状态机 FSM 驱动的。newChaincodeSupportHandler 状态机的初始化代码：

```
v.FSM = fsm.NewFSM(createdstate, fsm.Events{...}, fsm.Callbacks{...})
```

状态机 FSM 注册的状态事件有：

```
//fabric/protos/peer/chaincode_shim.pb.go中定义
//REGISTER即pb.ChaincodeMessage_REGISTER.String()对应的字符串值
//REGISTER事件表示从状态createdstate到状态establishedstate
REGISTER Src: []string{createdstate},      Dst: establishedstate
READY
PUT_STATE
DEL_STATE
INVOKE_CHAI
COMPLETED
```

```

GET_STATE
GET_STATE_B
GET_QUERY_R
GET_HISTORY
QUERY_STATE
QUERY_STATE
ERROR
RESPONSE
INIT
TRANSACTION
RESPONSE
INIT
TRANSACTION

```

状态机 FSM 涉及的事件状态有：

```

//在/fabric/core/chaincode/handler.go中以常量的形式定义
createdstate      = "created"
establishedstate  = "established"
readystate       = "ready"
endstate         = "end"

```

状态机 FSM 状态事件调用的时机函数为：

```

//在REGISTER事件发生之前调用beforeRegisterEvent
"before_REGISTER"      : beforeRegisterEvent
"before_COMPLETED"     : beforeCompletedEvent
"after_GET_STATE"       : afterGetState
"after_GET_STATE_BY_RANGE" : afterGetStateByRange
"after_GET_QUERY_RESULT" : afterGetQueryResult
"after_GET_HISTORY_FOR_KEY" : afterGetHistoryForKey
"after_QUERY_STATE_NEXT" : afterQueryStateNext
"after_QUERY_STATE_CLOSE" : afterQueryStateClose
"after_PUT_STATE"       : enterBusyState
"after_DEL_STATE"       : enterBusyState
"after_INVOKE_CHAINCODE" : enterBusyState
//表示在进入established状态之时调用enterEstablishedState
"enter_established"     : enterEstablishedState
"enter_ready"          : enterReadyState
"enter_end"            : enterEndState

```

在 HandleMessage 函数中，对传入的数据 msg 简单验证后，`eventErr := handler.FSM.Event(msg.Type.String(), msg)` 触发了状态机的状态事件，进而触发了对应的事件函数。

以“REGISTER 类型的 ChaincodeMessage”为例。客户端通过 grpc 发送 REGISTER 类型的 ChaincodeMessage 信息，服务端通过 msgAvil 频道接收后传入 HandlerMessage 函数，状态机对应执行 REGISTER 状态事件，状态从 createdstate 转变为 establishedstate，同时在转变之前自动触发 beforeRegisterEvent 函数完成注册。当状态进入 establishedstate 时，又接着触发了“enter_established”所对应的 enterEstablishedState 时机函数去通知客户端注

册已经正确完成。

在 `beforeRegisterEvent` 函数中, 通过 `err = handler.chaincodeSupport.registerHandler(handler)` 完成注册, 使用的是创建 `Handler` 时传入进来的 `ChaincodeSupport` 对象的 `registerHandler` 函数。注册过程就是将 `Handler` 对象赋值给 `ChaincodeSupport` 对象的 `runningChaincodes` 中的 `chaincodeMap` 映射: `chainID` 作为 `key`, 以 `Handler` 对象为成员 `handler` 值的 `chaincodeRTEnv` 对象作 `value`。

6.2.8 serialSend 或 serialSendAsync

两者都是使用 `Handler` 中 `grpc` 服务端流接口 `ChatStream` 成员发送 `ChaincodeMessage` 消息的函数, 两者都将应答 `ChaincodeMessage` 信息发送给客户端, 也都实现了将所发送的 `ChaincodeMessage` 信息串行化的目的。区别在于 `serialSend` 是阻塞发送, 而 `serialSendAsync` 是利用新启 `goroutine` 进行非阻塞发送, 且这些非阻塞的 `goroutine` 中任何一个发生发送消息的错误, 都会利用 `errc` 频道将错误发送给 `processStream` 函数。

6.2.9 系统 chaincode

系统 `chaincode` 与普通 `chaincode` 的编程模型相同, 只不过它运行于 `peer` 节点内而非一个隔离的容器中。因此, 系统 `chaincode` 在节点内构建且不遵循上文描述的 `chaincode` 生命周期。特别地, 安装、实例化、升级这三项操作不适用于系统 `chaincode`。

系统 `chaincode` 的目的是削减 `peer` 节点和 `chaincode` 之间的 `grpc` 通讯成本, 并提升了管理的灵活性。例如: 一个系统 `chaincode` 只能通过 `peer` 节点的二进制文件升级。同时, 系统 `chaincode` 只能以一组编译好的特定的参数进行注册, 且不具有背书策略相关功能。

系统 `chaincode` 在 `Hyperledger Fabric` 中用于实现一些系统行为, 所以可以被系统开发者适当替换或更改。

以下是系统 `chaincode` 的列表:

- ❑ `LSCC`: 生命周期系统 `chaincode` 处理上述生命周期相关的功能。
- ❑ `CSCC`: 配置系统 `chaincode` 处理 `peer` 侧 `channel` 的配置。
- ❑ `QSCC`: 查询系统 `chaincode` 提供账本查询 `API`, 比如获取区块及交易等。
- ❑ `ESCC`: 背书系统 `chaincode` 通过对交易响应进行签名来处理背书过程。
- ❑ `VSCC`: 验证系统 `chaincode` 处理交易的验证, 包括检查背书策略以及多版本并发控制。

替换或更改这些系统 `chaincode` 一定要小心, 尤其是 `LSCC`、`ESCC` 和 `VSCC`, 因为它们处于主交易执行路径中。值得注意的是, `VSCC` 在一个区块被提交到账本之前进行验证, 故所有 `channel` 中的 `peer` 节点得出相同的验证结果以避免账本分叉 (不确定因素) 就很重要了。所以当 `VSCC` 被更改或替换时要更加小心。

6.3 chaincode 数据结构分析

6.3.1 chaincode 元数据

chaincode 元数据包含以下几类：

1) 命令行 Flag。在 `peer/chaincode/chaincode.go` 的 `init()` 中调用 `resetFlags()` 初始化 flagset，其中包括 `chaincodeLang`、`chaincodeCtorJSON`、`chaincodePath`、`chaincodeName` 等，并在 `chaincode` 的每个子命令初始化时调用 `attachFlags` 添加子命令需要的 Flag。

2) policy。指明 chaincode 所用到的策略，其中包含 chaincode 部署时使其生效的签名者。策略的字符串格式为 `"AND('A.member', 'B.member'), OR('C.admin', 'D.member')"` 并且支持 `"OR(AND(X, Y), AND())"` 格式的嵌套结构。其中 (X,Y) 代表 `ORG.ROLE(A.member)`，ORG 表示一个组织 MSP 的 ID，ROLE 表示该 MSP 所管理的成员角色，包括 `member` 和 `admin`。并由 `common/cauthdsl/policyparser.go` 中的 `FromString()` 函数对策略字符串进行解析，该函数使用了第三方库 `govaluate` 解析策略字符串并存储到一个 `SignaturePolicyEnvelope` 对象中，为了存储策略字符串中的嵌套结构，该对象中的成员 `SignaturePolicy` 采用的是递归结构。`SignaturePolicy` 包含 `SignaturePolicy_SignedBy` 和 `SignaturePolicy_NOutOf_` 两种类型的对象，`SignaturePolicy_NOutOf_` 是递归结构的，有成员 `SignaturePolicy` 和成员 `N`，成员 `SignaturePolicy` 用于嵌套下一层策略的字符串，成员 `N` 表示当前嵌套层的策略是 AND 还是 OR，1 表示 OR，2 表示 AND。最终 policy 会将调用 `protos/utis/commonutis.go` 中的 `proto.Marshal()` 存储到 `chaincode.go` 的变量 `policyMarhsalled` 中。

3) `chaincodeCtorJSON`。指定 chaincode 要运行的函数和函数的参数，格式为 JSON 格式的字符串，如 `{ "Function": "func", "Args": ["param"] }`。可以直接调用 `json.Unmarshal`，将字符串存入一个 `ChaincodeInput` 对象中，作为 `ChaincodeSpec` 的 `Input`。

4) `chaincodeSpec`。在 `protos/peer/chaincode.pb.go` 中定义，描述 chaincode 的结构体，简称为 CS。它的成员有：`Type` 指定 chaincode 的语言类型，当前版本的 Fabric 只支持 Go 语言的 chaincode；`ChaincodeId` 指定了 chaincode 的路径、名称、版本；`Input` 存储指定的 chaincode 的运行的函数和函数的参数。这些数据都来自于 Flag。

5) chaincode 部署说明书。`ChaincodeDeploymentSpec` 在 `protos/peer/chaincode.pb.go` 中定义，描述一个 chaincode 该如何部署，简称 CDS。成员有：`ChaincodeSpec`；`EffectiveDate` 记录了 chaincode 何时有效的时间戳，即在 chaincode 开始运行时记录下时间点；`CodePackage` 存储了一个 `.tar.gz` 压缩包的二进制数据，这个压缩包包含 chaincode 源码以及源码所依赖的第三方库；`ExecEnv` 标识 chaincode 的运行的环境，表明是运行在 Docker 容器中还是操作系统之中。其中 `CodePackage` 经 `core/container/vm.go` 中的 `GetChaincodePackageBytes()`，选择 Go 平台的 `goPlatform` 对象后，最终调用 `core/`

chaincode/platforms/golang/platform.go 中的 GetDeploymentPayload, 根据 chaincode 说明书中记录的 chaincode 信息, 生成压缩包。这个过程的复杂之处主要在于收集 chaincode 源码所依赖的第三方库, 在搜集过程中, 排除掉了 GOROOT 和 Fabric 项目已经提供的库, 也排除掉了 chaincode 源码目录路径中所有的 vendor 目录中的库, 剩下的依赖的第三方库都被重新映射到 chaincode 源码目录下的 vendor 目录。最终, 将源码文件和所依赖的库都打进一个 .tar.gz 压缩包中并返回。如 chaincode 源码的路径是 GOPATH/src/hyperledger/fabric/examples/chaincode/go/chaincode_example02/, 假设该 chaincode 源码依赖第三方库 github.com/jmoiron/sqlx (放在 GOPATH/src/ 下, 实际上不依赖), 则生成的压缩包中的路径为 src/hyperledger/fabric/examples/chaincode/go/chaincode_example02/, chaincode_example02 目录下包含 chaincode 所有源码和一个 vendor 目录, vendor 目录中包含路径 github.com/jmoiron/sqlx/, sqlx 目录中包含 sqlx 库全部的文件。

6) ccpackfile 包文件。一种由 chaincode 命令的 package 或 signpackage 子命令生成的 chaincode 二进制部署包。前者是由 CDS 对象生成, 后者是由 Envelope 对象 (包含签名过的 CDS) 生成。将这两者形成的 ccpackfile 使用 ioutil.ReadFile 读入一个 buf 中后, 可以使用 CDSPackage 对象或 SignedCDSPackage 对象的 InitFromBuffer 先将 buf 中的数据转入对象, 然后调用对象的 GetPackageObject 从对象中抽取部署数据, 接着将其尝试转化为 CDS 或 Envelope 供部署时使用。我们将 CDS 和用于部署的 Envelope 都称作部署数据。

7) CDSPackage/SignedCDSPackage 对象。在 core/common/ccprovider/ 下定义, 也是一种存储 chaincode 部署说明书数据的对象, 但同时也提供了一系列供不同情况下调用的接口, 实际上是作为将 CDS 转化为其他所需要格式的中间件。

8) ChaincodeInvocationSpec。chaincode 调用说明书, 简称为 CIS。主要存储了一份 chaincode 说明书, 只不过这份说明书可能是关于某一个 systemchaincode 的说明书, Input 中存储的函数变为 install/upgrade 等对 ACC 进行操作的函数, 而用户的部署数据 (CDS 或 Envelope) 则变为该函数的参数。比如用户要安装一个自己的 chaincode, 生成的调用说明书中, 会使用到一份系统链 lsccl 的说明书, 而对于 lsccl 说明书中的输入 Input 来说, 操作的函数为 install, 函数的参数为用户 chaincode 的部署数据。

9) Proposal/SignedProposal。Proposal 封装了 chaincode 的数据, 如部署包等。作为一个申请消息, 让节点签名后, 会同签名数据一同放入 SignedProposal, 而 SignedProposal 就是 chaincode 可以通过 grpc 与节点进行通信的数据包结构, 即 ACC 向节点发送执行交易的消息, 都是以 SignedProposal 消息的形式发送出去的。

10) CCContext。字面意思就是 chaincode context, 即链码内容。也是描述 chaincode 自身信息的一种载体, 记录了 chaincode 的一些关键信息, 如成员 ChainID 指定了代表的 chaincode 所在的链的 ID, 成员 Name 指定了代表的 chaincode 的名字, 成员 Syscc 指定了代表的 chaincode 是否是 SCC。

11) transactionContext。交易上下文，在交易中产生，以交易 id 为 key，记录在 Handler 的一个 map 中，随用随删。比如一个部署交易中，在 Handler 处理这个交易期间，会产生一个这样的交易上下文，存储关于这个交易的一些信息，供处理交易的主体使用。待该次部署交易完成，则会将其删除。

12) Chaincode 消息。ChaincodeMessage，链对象的服务端和 shim 端进行消息交互的主要消息载体，在 protos/peer/chaincode_shim.pb.go 中定义。成员 Type 指定了消息的类型，有 ChaincodeMessage_REGISTER、ChaincodeMessage_INIT 等。成员 Txid 指定了该消息所在的交易编号（每个 chaincode 执行的交易都会有一个编号，每个交易会使用到多个类型的 ChaincodeMessage 在 chaincode 的服务端和 shim 端进行交互）。成员 Timestamp 自然是时间戳。成员 Payload、Proposal 是消息承载的 chaincode 数据，如源码包、执行的参数等（因为是 []byte 格式的所以内容类型很自由），并且可以为空。成员 ChaincodeEvent 是 chaincode 在 Init 或 Invoke 时给回的所要触发的事件，比如，在 chaincode 部署时，当部署完毕后，shim 可能给回服务端一个 Event 让服务端去触发，做一些其他必要的事情，同样可以为空。

13) CCPackage/SignedCDSPackage。分别定义在 core/common/ccprovider/ 下的 cdspackage.go 和 sigcdspackage.go 中，两者都实现 ccprovider.go 中定义的 CCPackage 接口，分别可以从一个 Marshal 过的 CDS 或 Envelope 抽取出其中所承载的关于 ACC 的数据来初始化自身，以便 SCC 检查 ACC 的数据是否符合要求，最终也是以这种两类结构写入文件系统来保存 ACC 的。

14) ChaincodeData。也是承载 ACC 数据的，是 ACC 在系统中部署时和最终向账本提交安装的数据格式。

15) StartImageReq。容器相关的数据结构，在 core/container/controller.go 中定义。包含了部署一个 chaincode 时启动一个容器需要知道的各种数据，如环境变量、网络 ID、节点 ID、部署包、构造函数、执行函数等。相应的有 StopImageReq、DestroyImageReq 两个结构体。

16) ChaincodeStub。相当重要的一个结构体，接口定义在 core/chaincode/shim/interfaces.go，在 chaincode.go 中实现。该结构体同样可以自己实现，参见 core/chaincode/shim/mockstub.go 和 mockstub_test.go。这个结构体是链码执行 Init、Invoke 两个接口时直接使用的数据，即描述一个交易具体是做什么的有关信息都存储在这里。

6.3.2 chaincode 的元工具

chaincode 的元工具介绍如下：

1) Signer。签名者，由 peer/chaincode/common.go 中的 InitCmdFactory 初始化。使用的是一个节点的本地 MSP 服务对象 bccspmsp 中的 signer，即一个 SigningIdentity。签名者的作用就是调用 CreateInstallProposalFromCDS 和 GetSignedProposal

对 chaincode 的部署数据签名 (CDS 或 Envelope), 形成一个已签名的申请 SignedProposal。

2) EndorserClient。背书客户端, 由 peer/chaincode/common.go 中的 InitCmdFactory 初始化。该服务端会随着一个节点的 start.go 的 serve 函数运行起来。背书客户端的作用是调用 ProcessProposal, 以此将签名者生成的 SignedProposal 发送给背书服务端执行交易后背书, 获得一个申请应答 ProposalResponse。ACC 的交易, 都是通过它来发起的。

3) BroadcastClient。一个连接 orderer 服务的广播客户端。chaincode 用其给 orderer 节点发送交易的结果数据, 供其进一步处理。

4) ccprovider。chaincode 提供者, 接口在 core/common/ccprovider/ccprovider.go 中的 ChaincodeProvider。唯一的实现是 core/chaincode/ccproviderimpl.go 中的 ccProviderImpl (ccprovider.go 中初始化的也是这个实例, 参看 ccproviderimpl.go 的 init())。这个工具相当于一个封装层, 置于 chaincode 与各种结构的 chaincode 数据之间。比较明显的就是在这个接口的方法中, 很多参数都是 interface{} 类型。即各种目的不同的 chaincode 数据都要经过它, 去往各个适合的地方。

5) ChaincodeSupport。chaincode 支持者, 是一个全局单例, 在 core/chaincode/chaincode_support.go 中实现和定义。为整个 chaincode 的框架提供支持, 更精确地说, 是提供 chaincode 整个服务端的框架支持。存储并引导执行所有 chaincode 所涉及的动作, 并不真正参与 chaincode 的执行。对于 ChaincodeSupport 来说, chaincode 需要的, 它都将尽量提供, 比如一个账本对象, 但之后不再干涉 chaincode 对账本对象的后续处理。当 chaincode 需要部署时, 会将 chaincode 交给其所属的 handler, 并等待结果且不干涉中间过程。如果 chaincode 需要某一类型的容器, 它会根据 chaincode 的需要返回给 chaincode inproc 容器或 Docker 容器对象, 然后让容器对象直接与 chaincode 交互, 不干涉后续过程。

6) container。容器, 即 chaincode 最终寄居的地方。有两种, inproc 容器和 Docker 容器。

7) Handler。分为服务端的 Handler 和客户端的 Handler, 各自封装了一个 FSM 状态机, chaincode 的交易就是由这两个 Handler 的状态机驱动的, 相当于 chaincode 交易的控制器和发动机。

8) txsimulator。交易模拟器, 用来模拟交易发生, 其实就是在内存 (典型的 map) 中记录交易产生的数据。在一个 chaincode 交易进行后, 会将数据由交易模拟器记录一份, 然后再把数据交给 orderer 服务处理 (最终提交至账本)。类似的还有 historyQueryExecutor 这类的查询工具。

6.4 SystemChaincode 讲解

SystemChaincode 为 Fabric 内部的系统链码, 区别于应用链码, 为 Fabric 工作的必要系统组件。

6.4.1 SystemChaincode

系统 chaincode 与普通 chaincode 的编程模型相同，只不过它运行于 peer 节点内而非一个隔离的容器中。因此，系统 chaincode 在节点内构建且不遵循上文描述的 chaincode 生命周期。特别地，安装、实例化、升级这三项操作不适用于系统 chaincode。

系统 chaincode 的目的是削减 peer 节点和 chaincode 之间的 grpc 通讯成本，并兼顾管理的灵活性。例如：一个系统 chaincode 只能通过 peer 节点的二进制文件升级。同时，系统 chaincode 只能以一组编译好的特定的参数进行注册，且不具有背书策略相关功能。

系统 chaincode 在 Hyperledger Fabric 中用于实现一些系统行为，故它们可以被系统开发者适当替换或更改。

在 start.go 中的 serve 函数里，在为 peerServer 注册 ChaincodeSupport 服务的函数 registerChaincodeSupport(peerServer.Server()) 中，注册了 System Chaincode：scc.RegisterSysCCs()。

SystemChaincode 的核心代码在 /fabric/core/common/sysccprovider 和 /fabric/core/scc 下，scc 也就是 SystemChaincode 的缩写。系统链码分为五种：cscs、escs、lscs、qscs、vscs，均为各个系统链码的缩写。系统链码均实现了 /fabric/core/chaincode/shim/interfaces.go 中定义的 chaincode 接口，所以 SystemChaincode 也属于 chaincode，只不过作用稍微特殊一点。

sysccprovider 目录下的文件有：

- sysccprovider.go，定义系统链码服务提供者接口

scc 目录下的文件有：

- sysccapi.go，系统链码的各种 api 操作。

- importsysccs.go，导入五种预定义的系统链码。

- sccproviderimpl.go，定义了系统链码服务提供者的具体实现和其操作。

6.4.2 预定义和注册

在 /fabric/core/scc/importsccs.go 中：

//预定义五个SystemChaincode存放到数组中

```
var systemChaincodes = []*SystemChaincode{
    {
        Enabled:      true,
        Name:          "cscs",
        Path:          "github.com/hyperledger/fabric/core/scc/cscs",
        InitArgs:      [][]byte{[]byte("")},
        Chaincode:      &cscs.PeerConfig{},
        InvokableExternal: true, // cscs is invoked to join a channel
    }, {...}, {...}, {...}, {...},
}
```

//注册五个系统链码

```
func RegisterSysCCs() {
    for _, sysCC := range systemChaincodes {
        RegisterSysCC(sysCC)
    }
}
```

RegisterSysCC 遍历 SystemChaincode 中所有的 SystemChaincode，并依次调用 Register SysCC 进行注册。RegisterSysCC 定义在 /fabric/core/scc/sysccapi.go:

```
//systemChaincode开启且处于白名单中
if !syscc.Enabled || !isWhitelisted(syscc) {...}
//最终将systemChaincode注册到系统中
err := inproccontroller.Register(syscc.Path, syscc.Chaincode)
```

inproccontroller.Register 定义在 /fabric/core/container/inproccontroller/inproccontroller.go:

```
//存放安装的chaincode, 以chaincode所在的path为key
typeRegistry = make(map[string]*inprocContainer)
//注册到typeRegistry
func Register(path string, cc shim.Chaincode) error {
    tmp := typeRegistry[path]
    if tmp != nil {
        return SysCCRegisteredErr(path)
    }
    typeRegistry[path] = &inprocContainer{chaincode: cc}
    return nil
}
```

Register 函数以 systemChaincode Path 成员值为 key，包含 systemChaincode 的 inproc Container 对象为 value，将系统链码放入 typeRegistry 映射中。至此，系统链码注册完毕。



注意 SystemChaincode 与一般 chaincode 一样，有相同的编程模型，不过 SystemChaincode 运行在 peer 程序中，是 peer 程序的一部分，而一般的 chaincode 是单独运行在一个容器中的。因此，SystemChaincode 是内建在 peer 程序中且不遵循一般 chaincode 那样的生命周期。install、instantiate 和 upgrade 操作也不适用于 SystemChaincode。

SystemChaincode 区别于普通 chaincode 的目的是减少 grpc 在 peer 节点与 chaincode 之间通信的时间消耗（因为 peer 节点在一个容器，chaincode 是单独的一个容器）。支持更灵活地管理，比如，一个 SystemChaincode 可以仅通过升级 peer 程序的二进制包来得到升级。SystemChaincode 可以用预定义的元素注册并编译到 peer 程序中，而且不需要有类似于背书策略或背书等这样的冗杂功能。

SystemChaincode 被用在 Fabric 中，去操纵整个系统的配置，这样的话可以随时把系统改变到合适的状态。

当前存在的 SystemChaincode 名单：

- ❑ LSCC, Lifecycle SystemChaincode, 处理生命周期请求。我理解的生命周期请求应该指的是一个 chaincode 的安装、实例化、升级、卸载等对其生命周期起关键作用的一系列操作请求。
- ❑ CSCC, Configuration System Chaincode, 处理在 peer 程序端的频道配置。
- ❑ QSCC, Query SystemChaincode, 提供账本查询接口, 如获取块和交易信息。
- ❑ ESCC, Endorsement SystemChaincode, 通过对交易申请的应答信息进行签名, 来提供背书功能。
- ❑ VSCC, Validation SystemChaincode, 处理交易校验, 包括检查背书策略和版本在并发时的控制。
- ❑ 在修改或替换 SystemChaincode 时必须注意, 特别是 LSCC、ESCC 和 VSCC, 因为它们处于重要的交易环节中。以 VSCC 为例, 因为区域链中的交易数据都是持久性的, 因此当 VSCC 在提交一个 block 到账本中之前先验证该块, 重要的是在同频道中的所有 peer 必须计算出相同的证书 (由验证输出的证书) 以避免账本产生冲突。因此特别需要注意的是 VSCC 被修改或替换时, 要避免和以前产生的交易数据产生冲突。

6.5 CSCC 分析

CSCC (Configuration SystemChaincode), 配置系统 chaincode 处理 peer 侧 channel 的配置。cscs chaincode configer 包提供了在网络重新配置时管理配置事务的功能。配置事务从 orderer 服务到达调用该 chaincode 的提交者。chaincode 还提供 peer 配置服务, 例如加入链或获取配置数据。

6.5.1 结构体

PeerConfiger 实现 peer 的配置处理程序。对于从 orderer 服务进入的每个配置事务, 提交者调用该 chaincode 来处理事务。它包含了一个策略检查器和一个配置管理器。

```
type PeerConfiger struct {
    policyChecker policy.PolicyChecker
    configMgr      config.Manager
}
```

6.5.2 函数

1. Init 方法

每个链在创建时调用一次 Init。这允许 chaincode 在链上的任何交易执行之前初始化账本上的任何变量。Init 函数通过 NewPolicyChecker 和 NewConfigSupport 创建了一

个策略检查器和一个配置管理器。策略检查器用于进行访问控制。

```
func (e *PeerConfiger) Init(stub shim.ChaincodeStubInterface) pb.Response {
    cnflogger.Info("Init CSCC")
    e.policyChecker = policy.NewPolicyChecker(
        peer.NewChannelPolicyManagerGetter(),
        mgmt.GetLocalMSP(),
        mgmt.NewLocalMSPPrincipalGetter(),
    )
    e.configMgr = peer.NewConfigSupport()
    return shim.Success(nil)
}
```

2. Invoke 方法

1) 处理加入链（由应用程序作为事务提议调用）。

2) 获取当前配置块（由 app 调用）。

3) 更新配置块（由提交者调用）Peer 使用 2 个参数调用该函数：

❑ args [0] 是函数名称，它必须是 JoinChain、GetConfigBlock 或 UpdateConfigBlock。

❑ args [1] 是一个配置块，前提是 args [0] 是 JoinChain 或 UpdateConfigBlock，否则它是链 ID。

4) validateConfigBlock 验证配置块，看它何时包含有效的配置事务。

5) joinChain 将加入配置块中的指定链。由于它是第一个块，因此它是包含此链的配置的生成块，因此我们想要使用此信息更新 Chain 对象。peer.CreateChainFromBlock 从块中创建一个链，peer.InitChain(chainID) 根据 chainID 初始化链，producer.CreateBlockEvents 创建块事件。然后用 producer.Send 方法发送事件，进行更新。

6) getConfigBlock 返回指定 chainID 的当前配置块。peer.GetCurrConfigBlock 用来获取当前配置块，如果 peer 不属于该链，则返回错误。

7) getConfigTree 方法返回指定 chainID 的当前 channel 和资源配置，通过 configMgr.GetChannelConfig 和 configMgr.GetResourceConfig 来获取 channel 配置和资源配置。如果 peer 不属于链，则返回错误。

8) 模拟配置树更新函数，通过 ProposeConfigUpdate 进行模拟配置更新。

9) 类型支持函数，根据不同的 channel 头部的类型返回不同的配置。

```
{
    switch common.HeaderType(channelHdr.Type) {
    case common.HeaderType_CONFIG_UPDATE:
        return pc.configMgr.GetChannelConfig(string(chainID)), nil
    case common.HeaderType_PEER_RESOURCE_UPDATE:
        return pc.configMgr.GetResourceConfig(string(chainID)), nil
    }
    return nil, errors.Errorf("invalid payload header type: %d", channelHdr.Type)
}
```

getChannels 函数返回关于该 peer 的所有 channel 的信息。每个 peer 可以存在于多个

channel 中。语句 `channelInfoArray := peer.GetChannelsInfo()` 可以获得一个 channel 的数组。

6.6 ESCC 分析

ESCC(Endorsement SystemChaincode): 背书系统 chaincode 通过对交易响应进行签名来处理背书过程。

6.6.1 结构体

EndorserOneValidSignature 实现了默认的背书策略, 即对提议哈希和读写集进行签名。

```
type EndorserOneValidSignature struct {
}
```

6.6.2 Init 函数

当 chaincode 第一次启动时, Init 被调用一次。

```
func (e *EndorserOneValidSignature) Init(stub shim.ChaincodeStubInterface)
pb.Response {
    logger.Infof("Successfully initialized ESCC")
    return shim.Success(nil)
}
```

Invoke 调用背书指定的提议。

现在, 我们签署输入并返回背书的结果。之后, 我们可以扩展 chaincode 以提供更复杂的策略处理, 例如将策略规范编码为 chaincode 的交易, 客户可以使用参数选择使用哪种策略进行背书。

- ❑ args [0], 函数名称 (现在不使用)。
- ❑ args [1], 序列化的 Header 对象。
- ❑ args [2], 序列化的 ChaincodeProposalPayload 对象。
- ❑ args [3], 执行 chaincode 的 ChaincodeID。
- ❑ args [4], 执行 chaincode 的结果。
- ❑ args [5], 仿真结果的二进制块。
- ❑ args [6], 序列化事件。
- ❑ args [7], payloadVisibility。



注意 这个链码是用来签署另一个链码的模拟结果。它不应该操纵状态, 因为任何状态变化都会被默默地抛弃: 如果这种背书是成功的, 那么将会持续的唯一状态变化就是我们将要签署的内容, 根据定义它不可能是我们自己的状态变化。

代码中把参数提取赋值给变量。通过 `utils.CreateProposalResponse(hdr, payl, response, results, events, ccid, visibility, signingEndorser)` 方法获取提案回复。`utils.GetProposalResponse(prBytes)` 收集提案响应以便我们返回其字节。

```
{
    ...
    hdr = args[1]
    ...
    payl = args[2]
    ...
    ccid, err := putils.UnmarshalChaincodeID(args[3])
    ...
    response, err := putils.GetResponse(args[4])
    ...
    results = args[5]
    ...
    if len(args) > 6 && args[6] != nil {
        events = args[6]
    }
    ...
    if len(args) > 7 {
        visibility = args[7]
    }
    ...
    //获取该peer的默认签名标识；它将用于签署此提议响应
    localMsp := mspmgmt.GetLocalMSP()
    ...
    signingEndorser, err := localMsp.GetDefaultSigningIdentity()
    ...
    //获取提案回复
    presp, err := utils.CreateProposalResponse(hdr, payl, response, results, events,
        ccid, visibility, signingEndorser)
    ...
    //收集提案响应以便我们返回其字节
    prBytes, err := utils.GetBytesProposalResponse(presp)
    ...
    pResp, err := utils.GetProposalResponse(prBytes)
}
```

6.7 LSCC 分析

LSCC(Lifecycle SystemChaincode)，生命周期系统 chaincode 处理与普通 chaincode 生命周期相关的请求。如安装、实例化、升级、卸载等。

生命周期系统 chaincode 管理这个 peer 上部署的 chaincodes。它通过 Invoke 提案管理链接代码 - “Args”: [“deploy”, <ChaincodeDeploymentSpec>] - “Args”: [“upgrade”, <ChaincodeDeploymentSpec>] - “Args”: [“stop”, <ChaincodeInvocationSpec>] - “Args”:

[“start”, < ChaincodeInvocationSpec>]

6.7.1 结构体和接口

具体实现代码如下：

```
// Support包含了LSCC执行任务所需要的函数，所谓的任务就是处理生命周期请求
type Support interface {
    // 把提供的chaincode包存储到本地。即文件系统
    PutChaincodeToLocalStorage(ccprovider.CCPackage) error

    // 为通过名字和版本指定的请求的chaincode取出chaincode包
    GetChaincodeFromLocalStorage(ccname string, ccversion string) (ccprovider.
        CCPackage, error)

    // 返回一个已经持久化到本地存储的所有chaincode数据的数组
    GetChaincodesFromLocalStorage() (*pb.ChaincodeQueryResponse, error)

    // 返回链接代码的实例化策略（如果没有指定，则返回通道的默认值）
    GetInstantiationPolicy(channel string, ccpack ccprovider.CCPackage) ([]byte, error)

    // 检查提供的签名提议是否符合提供的实例化策略
    CheckInstantiationPolicy(signedProposal *pb.SignedProposal, chainName string,
        instantiationPolicy []byte) error
}

// lifeCycleSysCC实现了chaincode生命周期和策略
type lifeCycleSysCC struct {
    // sccprovider是我们在没有导入周期的情况下调用系统chaincode包的方法的接口
    sccprovider sysccprovider.SystemChaincodeProvider

    // policyChecker是用来体现访问控制的接口
    policyChecker policy.PolicyChecker

    // 提供了几个静态函数的实现
    support Support
}

//创建LSCC实例
func NewLifeCycleSysCC() *lifeCycleSysCC {
    return &lifeCycleSysCC{support: &supportImpl{}}
}
```

6.7.2 函数操作

在给定的链上创建 chaincode：

1) putChaincodeData 方法添加 chaincode 的数据。在方法开始前检查了 ESCC 和 VSCC 是否为真实的系统 chaincode。

2) putChaincodeCollectionData 方法添加链码的集合数据，通过使用 collection ConfigBytes []byte 参数。其中将添加对集合对象的验证。LSCC 仅支持不可变的集

合，并将在后续版本引入对集合升级的支持。

3) getCCInstance 方法检查给定 channel 上是否存在 chaincode，方法试图通过 chaincode 名字获取 chaincode 的实例。

4) getChaincodeData 方法从字节中获取 chaincode。通过 cdbytes []byte 参数传入字节。

5) getCCCode 方法检查给定链上是否存在 chaincode。

6) getChaincodes 方法返回在此 LSCC 通道上实例化的所有 chaincode。其中定义一个数组 var ccInfoArray []*pb.ChaincodeInfo 来存储来自 LSCC 的所有 chaincode 的元数据。如果系统中没有安装 chaincode，我们将不会有名称和版本以外的数据。最后，将所有有关实例化 chaincodes 的信息添加到查询响应原型中。

7) getInstalledChaincodes 方法返回 peer 上安装的所有 chaincode。获取链码查询响应原型，其中包含所有有关已安装的 chaincode 的信息。

8) isValidChainName 方法检查链名的有效性。

9) isValidChaincodeName 检查 chaincode 名称的有效性。chaincode 名字不应该是空白的，应该只包含字母、数字、'_' 和 '-'。

10) isValidChaincodeVersion 用来检查链码版本的有效性。版本不应该是空白的，应该只包含字母、数字、'_'、'-'、'+' 和 '。

11) isValidCCNameOrVersion 方法验证 chaincode 名字或者版本的有效性。

12) isValidStatedbArtifactsTar 函数从存档中提取元数据文件，遍历文件并验证。

6.7.3 安装、部署和升级

下面讲解下 chaincode 的安装、部署和升级操作。

1) executeInstall 实现“安装”调用事务。首先验证 chaincode 的名称和版本，然后从 chaincode 包中获取任何陈述的工件，例如 couchdb 索引定义。cceventmgmt.GetMgr().HandleChaincodeInstall 启动安装。最后，如果完成了以上所有内容，通过 PutChaincodeToLocalStorage 将 chaincode 安装到本地 peer 文件系统，以便启动背书。

2) executeDeployOrUpgrade 根据其函数参数将代码路径路由到 executeDeploy 或 executeUpgrade。此处直接分析 xecuteDeploy 和 executeUpgrade 方法。

```
switch function {
    case DEPLOY:
        return lsc.executeDeploy(stub, chainname, cds, policy, escc, vsc, cd, ccpack,
            collectionConfigBytes)
    case UPGRADE:
        return lsc.executeUpgrade(stub, chainname, cds, policy, escc, vsc, cd, ccpack)
    default:
        logger.Panicf("Programming error, unexpected function '%s'", function)
        panic("") // unreachable code
}
```


3) `executeDeploy` 实现“实例化” `Invoke` 事务。首先通过 `getCCInstance` 获取 `chaincode` 实例, 测试 `LSCC` 中 `chaincode` 的存在。保留链码特定的数据并填写渠道特定的数据, 如 `cdfs.Escc`, `cdfs.Vscc`, `cdfs.Policy`。 `GetInstantiationPolicy` 获得实例化策略并通过 `CheckInstantiationPolicy` 对策略进行检索和评估。

4) `executeUpgrade` 实现了“升级”调用事务。只检查是否存在 `chaincode` 实例(它必须存在于通道上)我们不关心 `FS` 上的旧链码, 甚至用户有可能已经删除了它。 `getChaincodeData` 用来比较版本, 如果版本相同就不升级, 如果违反实例化策略也不要升级。以后几步和部署相同, 获取已签名的实例化建议, 保留链码特定的数据并填写渠道特定的数据, 检索和评估新的实例化策略。

6.7.4 chaincode stub 接口实现

`chaincode stub` 为链码程序提供用来操作 `Fabric` 统一的一层封装, 方便用户访问底层区块链。

1. Init

`Init` 初始化 `chaincode` 提供者, 为了实现访问控制也初始化了策略检查器。

```
func (lsc *lifeCycleSysCC) Init(stub shim.ChaincodeStubInterface) pb.Response {
    lsc.sccprovider = sysccprovider.GetSystemChaincodeProvider()
    // 初始策略检查器进行访问控制
    lsc.policyChecker = policyprovider.GetPolicyChecker()
    return shim.Success(nil)
}
```

2. Invoke

`Invoke` 操作有 `INSTALL`、`DEPLOY`(语义上的实例化)、`UPGRADE`、`GETCCINFO`、`GETDEPSPEC`、`GETCCDATA`、`GETCHAINCODES`、`GETINSTALLEDCHAINCODES`。下面列举对应的操作和参数:

❑ 安装

❑ 验证 `SignedProposal` 是不是 `admin` 节点签署的 (check 时传入的 `ADMINS`)。

❑ executeInstall()

❑ 检查 `cc` 名字和版本是否合法, 不包含非法字符。

❑ 把 `ccPackage` 写入文件系统。

❑ 部署

❑ 检查参数, 如果空则赋值默认值, `policy` 默认是被任意一个成员签署, `escc` 和 `vscc` 默认就是系统的 `escc` 和 `vscc`, 暂不支持自己指定, 因为后面会检测指定的 `cc` 是不是 `scc`, 而 `scc` 目前就是固定的这几个不能动态增加。

❑ executeDeploy()

○ 检查 `cc` 名字和版本是否合法, 不包含非法字符 (以及 `ACL`, `access control`, `TODO`)。

- 检查链码是否已经存在，就是已经被实例化。
- 获取 ccPackage 并转换成 ChaincodeData。
- 验证实例化策略，ccPackage 有两种，源码打包生成的，还有 install 时直接传入的，直接传入的包有可能是被签名的，被签名的 ccPackage 是被指定了实例化策略的，只有指定的身份才可以实例化这个 cc，其他所有的没被签名的 ccPackage 默认任意一个 ADMIN 节点都可以实例化。平常用的 install 命令，本地传入地址和参数打包生成的 ccPackage 都是未被签名的。
- createChaincode()，检查指定的 escc 和 vscc 是不是 scc（所以暂不支持自己的背书和验证链码），最后 putState()，其中 key 是 cc 名字，value 是序列化后的 ChaincodeData。
- 升级
- 跟 DEPLOY 类似，参考 DEPLOY，多了一步检查版本号是不是跟旧的版本号不同。
- GETCCINFO GETDEPSPEC GETCCDATA
- 检查通道 Readers 策略，是否有可读权限。
- 检查 cc 是否被实例化，成功则按照函数名返回相应信息。
- GETCCINFO 返回 cc 名（成功获取 ChaincodeData）。
- GETDEPSPEC 返回序列化后的 ChaincodeDeploymentSpec。
- GETCCDATA 返回序列化后的 ChaincodeData。
- GETCHAINCODES GETINSTALLEDCHAINCODES
- 检查是不是 ADMIN 节点。
- GETCHAINCODES 返回所有已经实例化的 cc（对 LSCC 的 state 进行范围查询）。
- GETINSTALLEDCHAINCODES 返回节点上所有安装的 cc（不一定实例化）。

6.8 QSCC 分析

QSCC(Querier System Chaincode)：查询系统 chaincode 提供账本查询 API，比如获取区块及交易等。获取区块或者交易有很多种方法，比如 etChainInfo、GetBlockByNumber、GetBlockByHash、GetTransactionByID。

6.8.1 结构体

LedgerQuerier 可以实现账本查询功能，包括：

- GetChainInfo 返回 BlockchainInfo。
- GetBlockByNumber 根据 number 返回一个块。
- GetBlockByHash 根据 hash 返回一个块。
- GetTransactionByID 返回一个交易。

```
type LedgerQuerier struct {
}
```

6.8.2 函数操作

1. Init 方法

每个链创建时都调用一次 Init。这允许链码在链上的任何事务执行之前初始化账本上的任何变量。

```
func (e *LedgerQuerier) Init(stub shim.ChaincodeStubInterface) pb.Response {
    qsccllogger.Info("Init QSCC")
    return shim.Success(nil)
}
```

2. Invoke 方法

1) 调用 Invoke 函数, args[0] 包含了查询函数名称, args[1] 包含链 ID, 它是暂时的, 直到它成为 stub 的一部分, arg[2] 接受查询的返回。通过 peer.GetLedger(cid) 方法获得账本。接下来处理访问控制列表, 第一步通过 stub.GetSignedProposal() 获得签署的提案, 第二步检查 channel 读策略。最后根据 fname 也就是查询函数名称来路由到对应的查询函数。每个函数都需要其他参数, 如下所述:

- ❑ GetChainInfo。返回按字节编组的 BlockchainInfo 对象。
 - ❑ GetBlockByNumber。在 args [2] 中返回块号指定的块。
 - ❑ GetBlockByHash。在 args [2] 中返回由块 hash 指定的块。
 - ❑ GetTransactionByID。在 args [2] 中返回由 ID 指定的交易。
- 2) getTransactionByID 通过交易 ID 来获取交易。
 - 3) getBlockByNumber 通过区块号来获取区块。
 - 4) getBlockByHash 通过 hash 值来获取区块。
 - 5) getChainInfo 获取区块链信息。
 - 6) getBlockByTxID 根据 TxID 获取区块。
 - 7) getACLResource 获取 ACL 资源。

6.8.3 路由规则

根据方法名做相应的路由, 进行不同方法的访问。

```
switch fname {
case GetTransactionByID:
    return getTransactionByID(targetLedger, args[2])
case GetBlockByNumber:
    return getBlockByNumber(targetLedger, args[2])
case GetBlockByHash:
    return getBlockByHash(targetLedger, args[2])
}
```



```

    case GetChainInfo:
        return getChainInfo(targetLedger)
    case GetBlockByTxID:
        return getBlockByTxID(targetLedger, args[2])
}

```

6.9 VSCC 分析

VSCC (Validator System Chaincode): 验证系统 chaincode 处理交易的验证, 包括检查背书策略以及多版本并发控制。

6.9.1 结构体

ValidatorOneValidSignature 实现了默认的交易验证策略, 该策略用于根据作为每个调用的参数提供的背书策略来检查读写集和背书签名的正确性。

```

type ValidatorOneValidSignature struct {
    //sccprovider是我们没有导入周期的情况下调用系统chaincode包的方法的接口
    sccprovider sysccprovider.SystemChaincodeProvider
    //collectionStore提供了从账本中检索集合的支持
    collectionStore privdata.CollectionStore
}

```

collectionStoreSupport 实现了 privdata.Support:

```

type collectionStoreSupport struct {
    sysccprovider.SystemChaincodeProvider
}

```

6.9.2 函数

Init 函数, chaincode 第一次启动时调用一次 Init, 给结构体中的变量赋初值。

```

func (vsc *ValidatorOneValidSignature) Init(stub shim.ChaincodeStubInterface)
pb.Response {
    vsc.sccprovider = sysccprovider.GetSystemChaincodeProvider()
    vsc.collectionStore = privdata.NewSimpleCollectionStore(&collectionStoreSup
        port{vsc.sccprovider})
    return shim.Success(nil)
}

```

1) Invoke 函数被调用来验证指定的交易区块。该验证系统链码将检查提供的信封中的交易是否包含符合提供的背书策略的背书 (即来自实体的签名)。peer 调用这个函数会附带三个参数: arg[0] 是函数名, arg[1] 是 Envelope, arg[2] 是验证策略。

Invoke 的步骤如下: GetEnvelopeFromBlock 从区块获取 envelope, GetPayload 获得 payload, GetApplicationConfig 接着获取应用配置, GetManagerForChain 获

取区块链 Manager, NewPolicyProvider 在之后创建策略提供者。验证 payload 类型, 获取交易, 根据策略评估签名集, 做一些 LSCC 相关特定的验证等。

2) checkInstantiationPolicy 根据签署的提案评估实例化策略。通过 chainName 获取 Chain 的管理器。创建一个策略提供者并创建一个策略。接着获取签名头部, 构建签名数据, 签名数据由 Data, Identity, Signature 组成。我们可以评估实例化的策略——Evaluate()。

3) validateDeployRWSetAndCollection 方法执行 LSCC 部署操作的 rwset 验证, 验证 rwset 时只能进行一次或者两次写操作。然后验证任何集合配置。这里一共进行了两次验证。

4) 验证 LSCC 调用, 根据传入的 LSCC 判断 LSCC 的生命周期, 根据 LSCC 路由到不同 case。进入升级和部署 case 会先获取 rwset, 为 LSCC 抽取 rwset, 接着从账本中检索拥有 chaincode 的入口。然后对 rwset 进行验证。如果具体是部署周期, 有三步安全检查, 再次验证 rwset, 检查实例化策略, 验证 cc 是否在实例化 cc 的 LSCC 表中。如果是升级周期, 首先也要验证 rwset, 接着验证 cc 是否在实例化 cc 的 LSCC 表中, 检查实例化策略, 检查现有的 cc 版本, 检查 rwset 中的实例化的策略。

5) getInstantiatedCC 方法获取实例化的 chaincode。首先获取账本的查询执行器——GetQueryExecutorForLedger。根据 chaincodeID 获取状态。接着通过 ccprovider.ChaincodeData 函数获取 chaincode 数据。

6) deduplicateIdentity 方法删除重复的身份。cap.Action.ProposalResponsePayload 作为签名信息的第一部分, 为评估创建签名, 遍历通过所有的背书——cap.Action.Endorsements, 并且最终创建签名集。遍历过程中已经添加了具有相同身份的背书, 设置签名的数据, 拼接了提案响应字节和背书者 ID, 设置签名信息的身份: 它是代言人, 最后设置签名。

6.10 SystemChaincode 的注册和实例化

SystemChaincode 系统链码是 Fabric 系统组件, 但是采用了灵活的方式, 可根据实际需求进行注册和实例化。

6.10.1 概述

SCC 的 Register 相当于 ACC 的 Install, SCC 的 Deploy 相当于 ACC 的 instantiate。只有这两个动作执行完毕, 一个 chaincode 才算真正可以使用。

SCC 启动占用的是 inproc 容器, 可以当作内存中的概念容器, 在 core/container/inproccontroller 下实现, ACC 启动占用的是 Docker 容器, 在 core/container/dockercontroller 下实现。

chaincode 接口是定义在 `core/chaincode/shim/interfaces.go` 中的 `chaincode`，只有两个接口：`Init(stub)` 和 `Invoke(stub)`，统一用同文件中的 `ChaincodeStubInterface` 接口实例作为唯一的参数。`ChaincodeStubInterface` 接口唯一的实现是在同目录下的 `chaincode.go` 中的 `ChaincodeStub`。

每一个 chaincode 都会带有两个 Handler，每一个 Handler 都是一个以状态机 (FSM) 驱动的通信机器，在驱动的过程中执行具体的状态事件，完成一个 chaincode 需要做的事情。一个可以称为服务端 Handler (相当于服务端) 实现在 `core/chaincode/handler.go` 中，另一个可以称为 shim (shim 本身有垫片的意思，可以理解为其是项目源码与开发者之间贴合的垫片) 端 Handler (相当于客户端)，在 `core/chaincode/shim/handler.go` 中实现。`core/chaincode` 下的部分为 chaincode 服务端的代码，主要用于处理 chaincode 的请求；`core/chaincode/shim` 下的部分为 chaincode 客户端的代码，用于定义供开发者使用的接口和客户端提交申请。

6.10.2 安装

SystemChaincode 的安装是在 `peer/node/start.go` 中进行的，所有的系统链码都是从这里进行具体的注册的。

在 `peer/node/start.go` 的 `serve` 函数中，调用了 `registerChaincodeSupport`，这其中执行了 `scc.RegisterSysCCs()`，针对每一个 SCC 依次执行 `core/scc/sysccapi.go` 中实现的 `RegisterSysCC(syscc)`，进行注册 (安装)。

`if !syscc.Enabled || !isWhitelisted(syscc)` 用来判断 SCC 是使能的且处于白名单之中，即 SCC 的 `Enabled` 要为 `true`，并且在配置文件 `core.yaml` 的 `chaincode.system` 项中，SCC 要配置为 `enable` (或 `yes/true`)。此时 SCC 的注册才能继续。

`inproccontroller.Register(lsccl.Path, lsccl.Chaincode)` 在 `core/container/inproccontroller` 处定义，注册 (即安装) SCC。上文说过，SCC 启动占用的是 `inproc` 容器，这里 `Register` 所做的事情就是把 SCC 的 chaincode 成员包装进一个 `inprocContainer` 容器，然后以 `scc.Path` 为 key 放进 `typeRegistry` 这个 map 中，这就算注册 (安装) 完毕了。从这一点就可以体会，相较于 ACC 的安装启动一个 Docker 容器，然后将 ACC 数据放入 Docker 容器内指定的目录，`inproc` 容器只是一个叫法上的容器。`typeRegistry` 这个 map 的作用也仅仅是存储注册的 SCC。

6.10.3 部署

在 `peer/node/start.go` 的 `serve` 函数中，调用了 `initSysCCs()`，这其中执行了 `scc.DeploySysCCs("")`，针对每一个 SCC 依次执行 `core/scc/sysccapi.go` 中实现的 `deploySysCC("", syscc)`，进行部署。这里需要留意第一个参数是空的，这个值是赋值给 `chainID` 的，`chainID` 的值对之后每一个 chaincode 的一系列操作有很大的影响，

也包括 SCC。与 chainID 类似，ChannelId、channel、chain 这类词指的都是链。每条链都有属于自己的管理范围，包括工具、账本等。只有属于这条链的 chaincode 才可以使用这条链上的工具并受其管理。SCC 不属于任何一个链，它属于 peer 节点，它只处理 ACC 的请求，ACC 在请求中提供 chainID，SCC 根据这个 chainID 从相应链上取出 ACC 需要的工具或数据返回给 ACC 使用。

`if !syscc.Enabled || !isWhitelisted(syscc)` 用来检测 scc 是否使能且处于白名单之中。`ctxt := context.Background()`，`if chainID != "" {...}` 会声明一个 Context，之后一直使用此变量。`if` 判断 chainID 是否为空，若不是空的，在 `if` 分支中所做的事情就是查看一下这条链上的账本和账本交易模拟器是否正常。即若 chainID 不为空，即指定了 SCC 部署到哪条链上，那一定是之后部署 scc 的时候会用到这条链的 Ledger 和 Ledger 的 TxSimulator。

`spec := &pb.ChaincodeSpec{...}`，`chaincodeDeploymentSpec`，`err := buildSysCC(ctxt, spec)`，`cccid := ccprov.GetCCContext(...)` 会根据原始的 scc 数据，一路封装，最后得到 scc 的部署包 CDS 和 CCContext，并将这些数据连同 ctxt 传入下一步，准备执行部署。

`ccprov.ExecuteWithErrorFilter(...)` 是 chaincode 执行交易的路线之一（另一个路线是 `core/chaincode/chaincodeexec.go` 中的 `ExecuteChaincode()`，ACC 执行这条线），经 `ccprovider` 导航，执行调用到 `core/chaincode/exectransaction.go` 中的 `ExecuteWithErrorFilter(...)`，SCC 的数据也一起传送至此，进而直接调用通文件中的 `Execute(...)`，到此才开始正式部署。



注意 所有的交易，无论是 ACC 的安装部署，查询，Invoke 等，最终都会到 `Execute(...)` 这里来开始，这里是交易产生和最终结束的地方。因此，要处理的情况更多，各种判断也会更多。

`Execute(...)` 接收的还是在前面封装的 SCC 数据，利用这些数据，主要执行了两步：`theChaincodeSupport.Launch()` 和 `theChaincodeSupport.Execute()`。在 `Execute` 执行完后，会对交易返回的应答消息 `resp` 进行判断，若成功，将返回 `resp` 和 `resp` 中的“倒钩事件”。

6.10.4 Launch

具体实现步骤如下：

1) 根据 scc 封装的数据，`cds` 不为空，`cID = cds.ChaincodeSpec.ChaincodeId`，`cMsg = cds.ChaincodeSpec.Input`，`canName := cccid.GetCanonicalName()`，从封装的数据中抽取出目前需要的数据之后，首先调用 `chaincodeSupport.chaincodeHasBeenLaunched(canName)` 进行判断 SCC 是否已经被 Launch 过。此函数的主要功能是

将 scc 以 canName (就是 chaincode 名字 + ' : ' + 版本号, 即 scc:1.0.0, 这是 chaincode 内部认可的权威名字) 为 key, 将 SCC 对应的 ServerHandler 放入这个 map 中。当然, 这里 SCC 是第一次部署, 其数据不会存在于 chaincodeMap 中, 所返回的 chrte 也自然为空。

2) if cds == nil { ... }, 该分支不会进入, 但是在 ACC 安装部署时会进入。

3) if (!....userRunsCC || cds.ExecEnv == ..) && (...) { ... } 对照 SCC 封装的数据会发现该分支可以进入。在分支中, 判断 if cds.CodePackage == nil 会进入, 但是 if !(....userRunsCC || cds.ExecEnv ==SYSTEM) 不会进入。直接到了 builder := func() ..., 定义了建立 Docker 容器的对象 builder, 并将 builder 作为参数之一传入 chaincodeSupport.launchAndWaitForRegister(...), 开始真正的 Launch 工作。这个 builder 对于 SCC 来说, 在之后不会使用, 对于 ACC 来说才会用到。这里罗列一下进入 launchAndWaitForRegister 的参数值: context、ccid、cds 仍是上文第 3, 4 封装的数据, builder 是此步产生, 但之后不会再用。

4) launchAndWaitForRegister(...) 是一个等待函数, 即一直要等到 Register 完毕之后才会返回。这里的 Register 指的是 scc 服务端 Handler 的状态, 即要等到 SCC 的 ServerHandler 处于 REGISTER 状态才会返回。在函数的开头, 再次调用了 chaincodeSupport.chaincodeHasBeenLaunched(canName) 查看 SCC 是否已经被 Launch 过。然后开始准备数据, 形成 ipCtxt、vmtype、sir (包含 CCID、Env、Args 等数据), 传入 container.VMCProcess() (core/container/controller.go 中定义), 并在其执行成功之后进入等待。这里有几点需要注意: a) 被包装进 sir 中的 preLaunchFunc 函数, 负责将 SCC 的 ServerHandler 生成并放入 chaincodeMap 中。(b) notify 是 ServerHandler 进入 REGISTER 状态的通知通道, 在 preLaunchFunc 中被赋值, 所赋的是 SCC 的 ServerHandler 实例中的 readyNotify 通道。而 select-case 等待的就是这个 notify 通道的通知, 也即 SCC 的 ServerHandler 实例中的 readyNotify 通道的通知。(c) ipCtxt 出自上文第 3 步创建的 ctxt 变量, 在这里向其中新增了 ChaincodeSupport 这个对象。

5) 在 VMCProcess() 中, 首先获取了 SCC 使用的 inproc 类型容器对象 v, 然后新起了一个 goroutine, 在锁保护下非阻塞的将 v 传入 StartImageReq (即上一步生成传入到这里的 sir) 的 do 函数, 然后使用 select-case 等待 do 函数的执行结束。

6) do 函数将 StartImageReq 自身已经初始化的每一个成员字段作为参数 (第 4 步), 依旧还有 ctxt, 传入 SCC 对应的 inproc 虚拟机对象 v 的 Start() 函数。这个 Start() 定义在 core/container/inproccontroller/inproccontroller.go 中, v 的原型为 InprocVM (Docker 容器的虚拟机对象原型为 DockerVM, 在 dockercontroller.go 中)。

7) 在 Start() 函数中, ipcTemplate := typeRegistry[path], ipc, err := vm.GetInstance(...) 所做的是先从 SCC 注册 (安装) 进 typeRegistry 的 map 第 3 步中取出 SCC 的 chaincode 实例, 作为新生成的 inprocContainer 容器的成员, 然后将这个属于 SCC 的容器放入 instRegistry 这个 map 中。if ipc.running 则检查 SCC 对应的容器 ipc 是否已经处于运行状态, 若重复部署 SCC 且能运行到这里, 那这里就会返回错误。ccSupport, ok

`:= ctxt.Value(...)` 则是从 `ctxt` 中取出之前放进去的 `ChaincodeSupport` (第 4 步 (c)), 继续执行 `prelaunchFunc()` (第 4 步 (a), (b)), 最后 `ipc.running = true`, 置 SCC 的 `inprocContainer` 为运行状态, 之后再对 SCC 进行 `chaincodeHasBeenLaunched(canName)` 检查返回都会是 `true`。`Start()` 函数最后在新启动的 `goroutine` 中去执行 `ipc.launchInProc`, 然后对 SCC 的两个 `Handler` 进行初始化。启动这个 `goroutine` 之后, `Start()` 返回, 结束了第 5 步中 `VMCProcess()` 的等待, 使得第 4 步的 `launchAndWaitForRegister(...)` 继续向下执行进入等待。

8) `launchInProc()` 分三部分, 启动两个 `goroutine`, 一个去启动 SCC 的 `ShimHandler`, 一个去启动 SCC 的 `ServerHandler`, 最后使用 `select-case` 等待两个 `Handler` 初始化完成。根据 `select-case` 等待结束的条件, 两个 `goroutine` 中任何一个结束 (无论成功与否), 都会导致函数结束返回。需要注意, 由于 SCC 是运行在 `inproc` 容器中的, 其实就是运行在内存中的, 所以 SCC 的 `ServerHandler` 与 `ShimHandler` 之间直接使用的是 `go` 中的 `chan` 进行交互, 即 `peerRcvCCSend` 和 `ccRcvPeerSend`, 并在 `core/container/inproccontroller/inprocstream.go` 中定义 `inProcStream` 来模拟 `grpc` 流 (只是模拟), 并实现了 `Send()` 和 `Recv()`, 这两个函数使用上述两个 `chan` 为两端的 `Handler` 收发消息 (`ServerHandler` 用 `peerRcvCCSend` 收, 用 `ccRcvPeerSend` 发, `ShimHandler` 则相反)。传入用于启动 `ShimHandler` 的函数 `shim.StartInProc()` 的参数: `env`、`args` 来自于第 4 步包含在 `sir` 中的数据, `ipc.chaincode` 则为第 7 步 `scc` 所部署的自身的 `chaincode` 对象实例。

9) 第一个 `goroutine`, 调用 `shim.StartInProc(...)`, 定义在 `/core/chaincode/shim/chaincode.go` 中。在这个函数中, `for _, v := range env` 首先从 `env` 中获取 SCC 的权威名字赋予 `chaincodename` (即 `scc:1.0.0`), `stream := newInProcStream(recv, send)` 再利用第 8 步提到的两个 `chan` 生成模拟收发流, 最后调用 `chatWithPeer(...)` 传入 `chaincodename`、`stream`、`chaincode` (即 SCC 的 `chaincode` 实例), 运行 `ShimHandler`。在 `chatWithPeer(...)` 中: (a) `handler := newChaincodeHandler(stream, cc)` 使用 `stream` 和 `chaincode` 创建了一个 `ShimHandler` 实例 `handler`。(b) `handler.serialSend(...)` 使用 `handler` 向 `ServerHandler` 发送一个 `ChaincodeMessage_REGISTER` 类型的 `ChaincodeMessage`, 即要服务端进入 `REGISTER` 状态的消息。注意, 由于第 8 步是连起两个 `goroutine` 去运行两个 `Handler`, 且工作量接近, 因此两个 `Handler` 运行起来的时间不会差太多。由于是通过 `chan` 发送的, 即便此时 `ServerHandler` 没有运行起来, `ShimHandler` 也会阻塞等待。(c) 发送完毕后, 新启一个 `goroutine` 在 `for` 循环中持续接收来自 `ServerHandler` 和自身状态机的消息, 然后 `chatWithPeer` 利用 `<-waitc` 等待该 `goroutine` 结束。在这个 `for` 循环中, 接收三个路径来的消息: 一是 `stream` 接收来自 `ServerHandler` 的消息后转发给 `msgAvail` 的消息, 二是自身状态机 `handler.nextState` 的状态消息, 三是 `stream` 发送消息失败的 `errc` 的消息。除 `errc` 通道接收到失败消息直接返回外, 另外两个路径来的消息都会交予 `ShimHandler` 处理, 即 `handler.handleMessage(in)`。倘若是 `handler`.

nextState 来的状态消息，还要再向 ServerHandler 发送，即 `else if nsInfo != nil && nsInfo.sendToCC` 分支中的内容。每次接收处理完一次消息后，for 循环将重置一些变量，再次进入等待接收三路消息的状态。

10) 第二个 goroutine, `newInProcStream(...)` 先使用第 8 步提到的两个 chan 生成了模拟收发流，再使用这个流调用 `ccSupport.HandleChaincodeStream(...)`，进而调用 `HandleChaincodeStream(...)` (定义在 `core/chaincode/handler.go` 中) 创建一个 ServerHandler 实例 (如果你够细心，或者阅读至此仍没有糊涂，你就会产生一个问题，即在前文的第 7 步执行的 `prelaunchFunc()` 函数，已经针对 LSCC 在 `chaincodeMap` 中注册了一个 Handler (也是 ServerHandler)，那为何这里又创建一个 ServerHandler 实例呢？这点下文会提及)。随即，执行 `handler.processStream()`，运行 ServerHandler。

11) 两端的 Handler 的初始状态都为 `createdstate` (创建状态)。ServerHandler 端的代码是这么安排的：接收和处理 (回复) 消息都在 `core/chaincode/handler.go` 中进行，接收用 `handler.processStream()`，处理 (回复) 用同文件下的各个 ServerHandler 的各个方法函数。ShimHandler 端的代码则是这么安排的：接收消息在 `core/chaincode/shim/chaincode.go` 中进行，处理 (回复) 消息则在同目录下的 `handler.go` 中进行。接收用 `chaincode.go` 中的 `chatWithPeer`，处理 (回复) 用 `handler.go` 中 ShimHandler 的各个方法函数。

12) ShimHandler 在启动前就向 ServerHandler 发送了一条 `ChaincodeMessage_REGISTER` 类型的 `ChaincodeMessage` 消息，ServerHandler 在启动后，于 `processStream()` 中接收到该条消息，交由 `handler.HandleMessage(in)` 处理，由于这条消息是 `REGISTER` 类型，则触发 ServerHandler 的状态机状态变为 `establishedstate` (建立状态)，同时触发了 `beforeRegisterEvent` 和 `enterEstablishedState` 这两个状态机的事件函数，且前者一定先执行完毕后会执行后者。(a) `beforeRegisterEvent` 主要做了两件事：第一，`handler.chaincodeSupport.registerHandler(handler)` 注册 LSCC 当前的 ServerHandler。第二，`handler.serialSend(...)` 向 ShimHandler 回复 `ChaincodeMessage_REGISTERED` 类型消息，注意这里加了 ED，即表示注册过的意思。第一件事就是第 4 步提及的等待 Register 完毕所要等的事情，也对应第 10 步所留下的疑问。该函数所做的就是先将之前 `processStream()` 预注册到 `chaincodeMap` 中的 Handler 的 `readyNotify` 通道赋值给当前的 ServerHandler 的 `readyNotify` 通道，然后用当前的 ServerHandler 替换这个预注册的 Handler，之后将 ServerHandler 的 `registered` 置为 true，给 `txCtxs`、`txidMap` 分配内存，这就是 Register 需要做的事情。(b) `enterEstablishedState` 主要做的通知的事情 (此时经过 (a) 的执行，ServerHandler 的 `readyNotify` 不可能为空)，就是调用 `handler.notifyDuringStartup(true)` 向 ServerHandler 的 `readyNotify` 通道发送一个 true 值。由于 (a) 中所做的替换的事情，发送后，步骤 7 中标记的还在 `select-case` 等待的 `launchAndWaitForRegister(...)` 会收到这个 true 值，立即结束等待并返回值为空

的 err, 即表明 ServerHandler 已经 Register 成功, 也即表明 ServerHandler 现在已经处于可以正常运行, 且收发处理消息的状态之中。

13) ShimHandler 收到第 12 步 (a) 发送来的 ChaincodeMessage_REGISTERED 类型消息, 交由 handler.handleMessage(in) 处理, ShimHandler 的状态机状态变为 established, 同时触发了 beforeRegistered 状态事件。beforeRegistered 函数没有继续做任何事情。

14) 在第 12 步 (b) 中 enterEstablishedState 执行之后, 第 4 步 launchAndWaitForRegister(...) 成功返回, Launch() 将于第 1、2、3、4 步后继续执行, 调用了 chaincodeSupport.sendReady(...), 目的在于由 ServerHandler 发起让自身和 ShimHandler 的状态机都进入 ready 状态, 没有做多余的事情。在 sendReady(...) 中: (a) if chrte, ok = chaincodeSupport.chaincodeHasBeenLaunched(...) 用于判断 SCC 于此时是否被 Launch 过, 注意这里用的是 !ok, 即此时 SCC 已经在 chaincodeMap 中被 Launch 了, 否则将错误返回。(b) notify, err = chrte.handler.ready(...) 让 scc 的 ServerHandler 发起进入 ready 状态的动作。(c) select-case 等待 notify 的通知, 即等待两端 Handler 都进入 ready 状态的通知。

15) 在第 14 步 (b) 的 handler.ready(...) 函数中, txctx, funcErr := handler.createTxContext(...) 创建了一个交易上下文对象的 transactionContext, 并将这个对象存储在 SCC 的 ServerHandler 的 txCtxs 这个 map 中, 以 txid (交易 ID) 为 key。txid 是部署之初生成的 txid, 部署也是一个交易, 而每个交易都会有唯一的 txid, 整个交易的过程也都使用这个 txid, 以此来避免交易的重复, 同时区分不同交易。这里创造的以 txid 为 key 的 transactionContext, 会在 ServerHandler 进入 ready 状态之后从 txCtxs 中被删除。生成了一个 ChaincodeMessage_READY 类型的 ChaincodeMessage 消息, 然后调用 handler.triggerNextStateSync(ccMsg), 把消息发送到 handler.nextState 后 handler.ready(...) 立即将 txid 对应的 transactionContext 的 responseNotifier 通道返回, 赋值给第 14 步 (b) 中的 notify, 并进入 (c) 中的等待。

16) processStream() 从 handler.nextState 中接收到 ChaincodeMessage_READY 类型的消息, 做了两件事: (a) 交由 handler.HandleMessage(in) 处理, 此 SCC 的 ServerHandler 根据此消息类型将状态机状态由 establishedstate 变为 readystate, 同时触发 enterReadyState 状态事件函数, 该事件函数只是通过 notify(msg), 将消息转发给取出的之前第 15 步存放在 txid (位于 txCtxs 中) 对应的 responseNotifier (位于 transactionContext) 通道 (这个通道经过第 15 步已经赋值给了第 14 步 (b) 中的 notify), 而现在第 14 步 (c) 中等待的 notify 收到这条消息后, 结束等待, 然后调用 handler.deleteTxContext(cccid.TxID) 将 txid 对应的 transactionContext 从 txCtxs 中删除, 第 14 步开始的 chaincodeSupport.sendReady(...) 结束返回, 继而 Launch() 结束。至此, SCC 的 ServerHandler 终于全部 Launch 结束。(b) 由于 handler.nextState 路径接收到的消息, 因此最后的 if nsInfo != nil && nsInfo.sendToCC 分支可以进入, 也就可以将 ChaincodeMessage_READY 消息异步

发给了 ShimHandler。

17) ShimHandler 的 chatWithPeer() 收到来自 ServerHandler 的 ChaincodeMessage_READY 消息, 交由 handler.handleMessage(in) 处理, ShimHandler 的状态机简单将状态从 established 变为 ready 后, 不会触发其他动作。至此, ShimHandler 端的工作在 Launch 阶段也彻底结束。

6.10.5 Execute

在执行完 theChaincodeSupport.Launch(...) 后, ccMsg, err = createCCMessage(...) 根据 Launch 返回的 CDS.CS.ChaincodeInput (实际为空) 和之前的 cctyp 生成的 ChaincodeMessage_INIT 类型的 ChaincodeMessage 传入 theChaincodeSupport.Execute(...), 进入 Execute 的参数: ctxt 和 cccid 仍是 SCC 部署之初生成的, ccMsg 是当前生成的, executetimeout 是 ChaincodeSupport 设置的超时时间。

1) if chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName) 会再次检查 scc:1.0.0 是否被 Launch 过, 若已被 Launch 则返回了 SCC 对应的 ServerHandler。到这一步, scc:1.0.0 必定已被 Launch。

2) notify, err = chrte.handler.sendExecuteMessage(...) 将利用 SCC 的 ServerHandler, 目的在于运行 ChainMessage 中所携带的数据指向的动作, 并返回通知通道给 notify。当顺利返回后, Execute(...) 函数将进入常规的 select-case 等待 (sendExecuteMessage 中的具体的任务依旧会异步执行)。

3) 在 sendExecuteMessage 函数中, handler.createTxContext(...), 依然用 txid 为 key 创建了一个交易上下文对象 transactionContext (这个对象会在 Execute 结束前调用 chrte.handler.deleteTxContext(msg.Txid) 被删除), 然后调用 handler.triggerNextState(msg, true) 将 ChaincodeMessage_INIT 类型的消息发送到 handler.nextState 通道, 然后将交易上下文对象的 responseNotifier 通道返回给 Execute 函数的 notify, 并让 Execute 函数进入等待。

4) ServerHandler 对于 handler.nextState 通道来的 ChaincodeMessage_INIT 类型消息不会触发任何状态的改变和事件函数。只会把该消息原封不动发给 ShimHandler。ShimHandler 在收到此消息后, 将触发 beforeInit 事件函数。beforeInit 事件函数中实际执行任务的是 handleInit(msg) 函数。

5) handleInit(msg) 函数是一个比较典型的处理流程。函数启动一个 goroutine, 在 goroutine 中, defer 执行最后要发送的消息 nextStateMsg (可能是正常的消息, 也可能是错误消息), 然后定义一个 errFun 函数, 一旦检测有错, 则返回一个 ChaincodeMessage_ERROR 类型的 ChaincodeMessage 消息 (即错误消息) 给 nextStateMsg, 随即 return, 也就触发了 defer。若没有发生错误, 则将一个 ChaincodeMessage_COMPLETED 类型的 ChaincodeMessage 赋值给 nextStateMsg, 函数结束, 触发 defer。handleInit(msg) 函数主要做了三件事: (a) stub := new(ChaincodeStub), stub.init(...) 组装

ChaincodeStub (对于 SCC 来说, 这个数据没有用, 但 ACC 会用到)。(b) `handler.cc.Init(stub)` 调用 SCC 的 chaincode 实例的 Init 接口, 然后初始化这个 chaincode, 也即 ChaincodeMessage_INIT 类型的消息就是指示 LSCC 做这个动作的。(c) 若成功 defer, 则发送 ChaincodeMessage_COMPLETED 类型消息。注意这里返回的 ChaincodeMessage 中给 ChaincodeEvent 成员赋值为 `stub.chaincodeEvent`, 这就是部署之初所提到的“倒钩事件”, 这个“倒钩事件”会随着 ChaincodeMessage 一同返回到 peer 节点最核心的地方。但是目前 SCC 和 ACC 都没有明显地使用这一点。

6) 第 5 步 (b) 处调用的是 LSCC 的 chaincode 实例的 Init 接口, 定义在 `core/scc/lscclsccl.go` 中, 所做的事情相当简单 (ACC 会稍微麻烦一些), 给 SCC 的两个成员赋值, 一个是 SCC 提供者实例, 一个是策略检查器。然后就返回成功标识 `shim.Success(nil)`。

7) 第 6 步执行完毕后, 第 5 步的 `handleInit(msg)` 也会随之结束触发 defer 而再向 ServerHandler 发送 ChaincodeMessage_COMPLETED 类型消息, ServerHandler 接收到此消息后状态机也会无动于衷, 只是在 `handler.HandleMessage(in)` 中会通过 `handler.notify(msg)` 将消息再转发给 txid 对应的 transactionContext 的 responseNotifier 通道。由此, 第 2 步 `Execute(...)` 函数的等待结束, 将收到的 ChaincodeMessage_COMPLETED 类型消息 `resp` 返回。

此时回看 `core/chaincode/exectransaction.go` 中的 `Execute(...)` 函数, 当 `theChaincodeSupport.Execute(...)` 执行完毕返回 ChaincodeMessage_COMPLETED 消息, 经过一些列的 if 判断, 会进入 `if resp.Type == pb.ChaincodeMessage_COMPLETED` 分支并成功返回。至此 `Execute(...)` 函数执行完毕, 转回同文件中的 `ExecuteWithErrorFilter(...)`, 亦是成功返回。一直回到最初的 `core/scc/sysccapi.go` 中的 `deploySysCC(...)`, 也是就此成功返回。就此, 整个 LSCC 的部署过程结束。

6.10.6 部署后状态

`theChaincodeSupport` 中的 `runningChaincodes.chaincodeMap` 中存在这以 `scc:1.0.0` 为 key 的 SCC 的 ServerHandler 实例。

SCC 的 ServerHandler 和 ShimHandler 均处于 ready 状态, 且接收发送消息的 goroutine 都在运行当中。

6.11 ApplicationChaincode 的部署

应用链码为实际应用根据特定的业务场景开发的智能合约, 用户需要将应用链码部署至需要背书该智能合约交易的节点上。

6.11.1 概述

`peer chaincode install` 命令执行安装命令，命令定义在 `peer/chaincode/install.go` 中，这是安装的起点。注意，此命令是在 `peer node start`、`peer channel create`、`peer channel join` 命令依次执行完毕之后所执行的，即执行 `install` 之时，`peer` 节点的基本模块（包括 SCC）都已初始化完毕，`channel` 也已经建立。

根据实例化的原则，从 `install_test.go` 中提取了一句实际的 `install` 命令：`peer chaincode install -n example02 -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v anotherversion`。安装的 ACC 是 `example02`，存在于 `-p` 指定的路径下，版本是 `anotherversion`。`-l` 没有，取默认的 Go 语言。

`install` 安装使用的 `chaincode` 数据有两种形式，一种是 CDS，一种是 `chaincode package/signpackage` 命令形成的 `ccpackfile` 包。因为我们没有涉及这两个命令，因此这里我们只以前一种 CDS 数据包为例，叙述安装过程。

`install` 能够识别的 flag 有 `-l`、`-c`、`-p`、`-n`、`-v`。其中 `-c` 不常用，其指定的是 ACC 所要执行的函数和函数的参数，一般在具体执行的时候，如查询或转账的时候再给定。

`install` 最终所要做的事情，就是将 `example02` 的源码包放入 Docker 容器的安装目录中。

6.11.2 生成签名申请包

以 `peer/chaincode/install.go` 的 `chaincodeInstall(...)` 为起点（设定 `ccpackfile==""`）。需要注意以下几点：

1) `chainID` 在 `SignedProposal` 中为空，即安装所要做的仅仅是把 `example02` 的压缩包放入容器的指定目录中而已，只有部署的时候才需要指定部署到哪条链上。

2) CDS 的 `CodePackage` 是 `example02` 的代码压缩包，包括源码和依赖的第三方库。因为 SCC 的一切都在项目编译时编进 `peer` 中了，所以 SCC 的 CDS 的这个字段就是空的，而 ACC 需要安装源码，所以这个字段在正常的情况下不是空的。这个压缩包最终是在 `core/chaincode/platforms/golang/platform.go` 中的 `GetDeploymentPayload()` 打包的。不仅需要打包源码还需要包含一直以来所有的包，这么做的目的就是让一个 `chaincode` 无论放到哪个容器里，不会因为缺少某个第三方库而编译失败。同时这个打包的过程也要求我们在执行 `example02` 安装的时候，要实现把其依赖的第三方库事先放到 `GOPATH/src` 下。

3) `txid` 是在 `/protos/utills/proputils.go` 中的 `CreateChaincodeProposalWithTransient(...)` 中计算出来的，是哈希（`peer` 节点的 MSPID+证书元数据+随机数 `nonce`）的值，可以把它当作一个唯一的字符串。

6.11.3 处理安装申请

在客户端将安装链码的请求发送到背书节点时，背书节点收到相应的请求会做出一系列的处理。

1) Endorser 服务端在 `core/endorser/endorser.go` 中的 `ProcessProposal(...)` 处, 接收到来自 Endorser 客户端发送的 `SignedProposal` 和一个之后一直会用到的 `Context` 上下文 `ctxt`。

2) 在 `ProcessProposal(...)` 中, 从开始至 `var txsim ledger.TxSimulator` 处, 其上的代码全部是一边解压抽取 `SignedProposal` 中的数据, 一边验证这些数据。

3) `var txsim ledger.TxSimulator` 和 `var historyQueryExecutor ...` 一个是交易模拟工具, 一个是历史查询执行工具。由于是 `Install`, 且 `chainID` 为空, 这两个值在之后都一直为空。`if chainID != ""` 的分支也不会进入 (当部署 `example02` 时, `chainID` 不为空, 则会进入此分支, 根据 `chainID` 获取这两个工具, 供之后部署使用)。

4) `ProcessProposal(...)` 所做的主要有两件事: (a) `e.simulateProposal(...)`, 模拟执行申请。(b) `e.endorseProposal(...)`, 背书申请执行的结果。但是由于 `chainID` 为空, 所以 `install` 命令不会执行此步 (同样, 部署时会用到), 而是直接以 `ProposalResponse` 的形式返回 (a) 中执行的结果。

6.11.4 执行申请

客户端发起交易背书提案至 `peer` 节点, `peer` 节点进行提案的一系列校验, 并模拟执行交易。

1) 传入 `e.simulateProposal(...)` 的参数: `ctx` 为上文所述的上下文 `ctxt` (至此未有更新), `chainID` 为空, `txid` 为交易 ID, `signedProp` 是客户端发送来的原数据, `prop` 是从 `signedProp` 中抽取出来的 `Proposal`, `hdrExt.ChaincodeId` 也是抽取出来的数据 (只包含一个值为 `LSCC` 的 `Name` 字段), `txsim` 为空。

2) 在 `e.simulateProposal(...)` 中, 前期又做了些简单的抽取和检查: `cis, err := putils.GetChaincodeInvocationSpec(prop)` 从 `prop` 中抽取出 `CIS`。`if err = e.disableJavaCCInst(cid, cis); err != nil` 通过判断 `example02` 的 `CDS.CS.Type` 来断定要安装的是否为 `Java` 源码, 当前版本不支持 `Java` 写的 `chaincode`, 但这部分在将来会被移除。`if e.checkEsccAndVscv(prop); err != nil` 为 `ESCC` 和 `VSCC` 对 `prop` 的检查, 但是当前版本未做什么实际的检查, 以后的版本有可能会加入。`if !syscc.IsSysCC(cid.Name){...}else{ version = util.GetSysCCVersion() }`, 由于 `cid.Name` 就是 `LSCC`, 因此只会进入 `else` 分支, 得到的为 `LSCC` 的版本值为 `1.0.0`。

3) 最后一步调用了 `e.callChaincode(...)`, 执行 `CIS` 指定的动作。上文所述的另一个函数 `endorseProposal()` 最后也是调用这个函数才开始执行申请任务的。进入 `e.callChaincode(...)` 的参数: `ctx` 依旧为 `ctxt`, `chainID`=空, `version`=`1.0.0`, `txid` 为交易 ID, `signedProp/prop/cid/txsim` 不变, `cis` 是第 2 步新抽取出的 `CIS`。

4) `callChaincode()` 函数做了三件事: (a) `cccid := ccprovider.NewCCCon`

`text(...)`, 根据传入的参数, 创建一个 `CCContext` 对象供执行申请所用。(b) `chaincode.ExecuteChaincode(...)`, 执行申请。(c) `if cid.Name == "lsccl" && len(cis.ChaincodeSpec.Input.Args) >= 3 && ...`, 这一步只有部署、升级的交易才会进入此分支。

5) `ExecuteChaincode(...)` 函数, 在 `core/chaincode/chaincodeexec.go` 中定义。`spec, err = createCIS(cccid.Name, args)` 根据传入的参数新建了一个 CIS, 其新生成的 CIS 和第 2 步从 `prop` 中抽取出的 CIS 在内容上是完全一致。接着就调用了 `Execute(ctxt, cccid, spec)`, 类似于 SCC。ACC 也就此进入了类似于 SCC 部署所述的机制和道路来进行 `example02` 的安装。只不过, 传入数据所承载的任务不同, 执行的方向也会稍有不同。

6) `Execute(ctxt, cccid, spec)` 仍依次执行 `theChaincodeSupport.Launch(...)` 和 `theChaincodeSupport.Execute(...)`, 但这里的 `spec` 是第 5 步生成的 CIS, 因此 `cctyp` 的值为 `ChaincodeMessage_TRANSACTION`, 进而生成的供 `theChaincodeSupport.Execute(...)` 使用的 `ccMsg` 是 `ChaincodeMessage_TRANSACTION` 类型的消息。

6.11.5 Launch

`Launch(...)` 函数在 ACC 安装的情况下执行不了太久, `canName := cccid.GetCanonicalName()` 等到的 `canName=lsccl:1.0.0`, 此值会使程序进入 `if chrte, ok = chaincodeSupport.chaincodeHasBeenLaunched(canName)`; 进入 `ok` 选择分支, 进而进入 `if chrte.handler.isRunning()` 分支而返回。

6.11.6 Execute

`Execute` 为具体的执行函数, 负责交易的具体模拟执行并返回响应。

1) 在 `Execute(...)` 函数中, `canName := cccid.GetCanonicalName()` 再次得到 `lsccl:1.0.0`, 然后通过 `chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName)`, 获取了 LSCC 的 `ServerHandler`, 在 `chrte.handler.sendExecuteMessage(...)` 开始使用该 `ServerHandler` 以触发状态机进入运行, 最后进入 `select-case` 等待。

2) 在 `sendExecuteMessage(...)` 中, 通过调用 `handler.triggerNextState(msg, true)`, `ServerHandler` 将 `msg` 发给自身的 `handler.nextState` 通道以触发 `ServerHandler` 的状态机进入下一个状态。

3) `ServerHandler` 的 `processStream()` 收到来自 `handler.nextState` 通道的 `msg`, 先交给 `handler.HandleMessage(in)` 处理, `ServerHandler` 状态机无任何变化, 然后 `handler.serialSendAsync(in, errc)` 给 LSCC 的 `ShimHandler` 发送 `msg`。

4) LSCC 的 ShimHandler 的 chatWithPeer() 收到 msg, 交由 handler.handleMessage(in) 处理, 触发 beforeTransaction 事件函数, 该事件函数主要调用同文件中的 handleTransaction(...) 函数。

5) handleTransaction(...) 函数主要做的就是根据 ShimHandler 收到的 msg 生成并初始化一个 ChaincodeStub, 对应图中的 ChaincodeStub, 然后 handler.cc.Invoke(stub) 调用 LSCC 的 Invoke() 方法对 example02 进行安装。

6) 在 LSCC 的 Invoke() 中, args := stub.GetArgs() 获取到的是 CIS.CS.Input.Args。这个数组的值来自于 protos/utills/propotills.go 中 createProposalFromCDS() 中的 case "install": 中的 ccinp。因此在 Invoke() 中, function := string(args[0]) 得到 function 的值是 "install", switch function 会进入 case INSTALL: 的分支。

7) 在 case INSTALL: 分支中, lsccl.policyChecker.CheckPolicyNoChannel(...) 首先专门使用了检测未指定 Channel 的 chaincode 函数来检查要安装的 example02。depSpec := args[1] 取出来的就是 example02 的 CDS, 不过此时的 CDS 仍是被 Marshal 过的。lsccl.executeInstall(stub, depSpec) 之后调用 LSCC 的函数, 依据 stub 和 example02 的 CDS, 执行安装。

8) 在 executeInstall(...) 中, 首先 ccpack,err := ccprovider.GetCCPackage(ccbytes) 会根据 example02 被 Marshal 过的 CDS 创建一个 CDSPackage (core/common/ccprovider/cdspackage.go 中定义)。其次, 简单验证 example02 的 Name 和 Version。最后, 调用 ccpack.PutChaincodeToFS() 将 example02 源码写入文件系统。

9) 在 PutChaincodeToFS(...) 中, 一系列 if 检查之后, 先 path := fmt.Sprintf("%s/%s.%s", chaincodeInstallPath, ccname, ccversion) 整合出要写入的路径, 即在 chaincodeInstallPath 目录下放入名为 example02.anotherversion 的文件。然后 os.Stat(path) 先检查此文件名是否可用。最后 ioutil.WriteFile(path, ccpack.buf, 0644) 将 CDSPackage 中成员 buf 写入 path 指定的地方。至此, example02 的安装申请执行完毕。由此, 开始一路返回。

10) 一路返回至第 6 步 ShimHandler 的事件函数 handleTransaction(...) 中, handler.cc.Invoke(stub) 返回, 继续向下执行, 将 nextStateMsg 赋值为 ChaincodeMessage_COMPLETED 类型的消息, 并执行 defer 中的 handler.triggerNextState(nextStateMsg, send), 并将该消息发送给自己的状态机。ShimHandler 只将 ChaincodeMessage_COMPLETED 消息发送给 ServerHandler 之后就再无其他动作或变化。

11) ServerHandler 收到 ChaincodeMessage_COMPLETED 消息, 通知仍处于等待之中的 Execute(...) 函数, 然后等待结束, Execute(...) 函数成功返回。

6.11.7 一路返回

当交易模拟执行完之后, 执行结果就会向客户端返回。

1) `Execute(...)` 函数结束之后, 就此一路返回, 一直返回到 `core/endorser/endorser.go` 中的 `callChaincode(...)`, `chaincode.ExecuteChaincode(...)` 执行完毕, 执行申请的 (6.11.4 节) 第 4 步的 (b), 由于这一步 (c) 中的 `install` 申请不会执行, 因此 `callChaincode(...)` 也就此结束。

2) 继续返回到 `simulateProposal`, 不会进入代码中的 `if txsim != nil` 分支, 因此也是直接返回至 `ProcessProposal()`。

3) 继续 `ProcessProposal()`, 将进入 `if res != nil` 分支, 但无法进入 `if res.Status >= shim.ERROR` 分支。因此继续向下走, 进入 `if chainID == ""` 分支对要返回给 Endorser 客户端的应答消息 `pResp` 赋值, 最后将 `pResp` 返回给 Endorser 客户端。

4) `example02` 为安装申请的起点, `peer/chaincode/install.go` 的 `chaincodeInstall(...)` 所调用的 `install(...)` 中的 `cf.EndorserClient.ProcessProposal()` 即是 Endorser 客户端, 收到服务端发来的消息在返回后, `install(...)` 随之结束, 进而导致 `chaincodeInstall(...)` 结束。至此, 整个 `example02` 的安装全部结束。

6.11.8 安装后的状态

在 `peer` 节点的 `chaincodeInstallPath` 目录下, 会有一个名为 `example02.anotherversion` 的文件, 该文件即为 `example02` 源码压缩包。用于实例化的时候创建相应链码的容器, 其中为链码对应的可执行文件的源码。

6.12 ApplicationChaincode 的实例化

链码安装之后需要进行实例化才能够进行后续调用, 实例化过程为创建链码的镜像, 并启动链码容器的过程。

6.12.1 概述

`peer chaincode instantiate` 命令执行部署命令, 命令定义在 `peer/chaincode/instantiate.go` 中, 这也是部署的起点。另外需要注意的一点是, 这个命令是在 `peer node start`、`peer channel create`、`peer channel join`、`peer chaincode install` 命令依次执行完毕之后执行的, 即执行 `instantiate` 之时, `peer` 节点的基本的模块 (包括 SCC) 都已初始化完毕, `channel` 已经建立, ACC 也已经安装。

根据实例化的原则, 从 `instantiate_test.go` 和官方文档中提取整合了 (尽量能用的 flag 都用上) 一句实际的 `instantiate` 命令: `peer chaincode instantiate -n example02 -v anotherversion -o orderer.example.com:7050 -C testchain -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.`

`member', 'Org2MSP.member')"`。-n 指定部署的 ACC 是 `example02`，-v 指定版本是 `anotherversion`，-o 指定连接的 orderer 服务实例的端点是 `orderer.example.com:7050`，-C 指定要部署的链是 `testchain`，-c 指定执行的函数和函数的参数，-P 指定策略。

`instantiate` 能够识别的 flag 有 -l、-c、-n、-v、-P、-E、-V、-C。相对于 `install`，这里就指定了 -c，要求部署的时候初始化 a、b 两个账户。

ACC 的部署是存储在 Docker 容器中的，启动的两端 Handler 通过 `grpc` 进行通信。

`instantiate` 最终要做三件事情，涉及三条链码：1) LSCC 执行部署交易将 `example02` 的源码放入自己的写集。2) `example` 执行自身的部署交易，启动 `example02` 容器并与 `peer` 节点通过 `grpc` 通信进行初始化，最后将初始化的状态写入自己的写集。3) 获取 LSCC，`example02` 的读写集，使用 ESCC 进行背书，然后将签名、读写集等部署产生的数据封装成 `Envelope`，发送到 `orderer` 节点交由其处理。

6.12.2 起点

起点位置在 `peer/chaincode/instantiate.go` 中的 `chaincodeDeploy(...)`，主要可以实现两件事：1) `env, err := instantiate(cmd, cf)` 与 `install` 命令执行的线路类似，对 `example02` 进行部署，并返回部署结果 `env`。2) `cf.BroadcastClient.Send(env)` 在部署成功的前提下，向各个节点广播部署结果 `env`。

6.12.3 部署

部署的步骤十分繁杂，下面将进行详细讲解。

1) `instantiate()` 是部署的起点，中途所组装的关于 `example02` 的 CDS 中，`CodePackage` 为 `nil`。`chainID` 值为 `testchain`，不再为空，即要把 `example02` 安装在 `testchain` 上。`CIS.CS.Input.Args` 的值依旧来自于 `protos/utlis/proputils.go` 的 `createProposalFromCDS()` 中的 `ccinp`，不过不同于 `install`，这次进入的是 `case "upgrade":` 分支（`case "deploy"` 中执行了 `fallthrough`），这点将直接影响 LSCC 的 `Invoke` 所进入的分支。

2) `cf.EndorserClient.ProcessProposal(...)` 将组装好的 `SignedProposal`，连同一个之后一直在用的 `Context` 上下文 `ctxt`，一起发给 `Endorser` 服务端。

3) `Endorser` 服务端在 `core/endorser/endorser.go` 中的 `ProcessProposal(...)` 接收到来自客户端的 `ctxt` 和 `SignedProposal`。`ProcessProposal(...)` 所做的事情依旧如 `install` 时所描述的那样，但不同于 `install` 之处在于，`chainID := chdr.ChannelId` 获取的值为 `testchain`，以至于之后三个 `if chainID != ""` 分支都会进入：(a) `lgr := peer.GetLedger(chainID)`，获取 `testchain` 的账本对象 `lgr`，并调用 `lgr.GetTransactionByID(txid)` 对 `txid`（交易 ID）的唯一性进行检查（这里是为了避免重复部署）。(b) 调用 `txsim, err = e.getTxSimulator(chainID)` 和 `historyQueryExecutor, err = e.getHistoryQueryExecutor`

(chainID);, 分别获取 testchain 的交易模拟工具和历史查询工具并赋值给 txsim 和 historyQueryExecutor, 同时将两个工具先后放入了 ctxt 中。(c) e.endorseProposal() 将会执行, 对 example02 的部署进行背书。

4) 在 e.simulateProposal(...) 中, 不同于 install 所述之处在于, 执行 e.callChaincode(...) 之后, 会进入 if txsim != nil 分支, 也会进入最后的 if cid.Name == "lsccl" && ... 分支。

5) 在 callChaincode(...) 中, 会进入 if txsim != nil 分支, 将第3步(b)中获取的交易模拟工具加入了 ctxt。

6) 在 chaincode.ExecuteChaincode(...) 中, 依旧生成 CIS 的对象, 与 ctxt、CCContext 一同传入同文件中的 Execute(...) 函数, 在这个函数中按部就班地开始 Launch 和 Excute。

7) 在此省略与 install 的 Launch/Excute 章节(6.11.5 节、6.11.6 节)相似的过程, 一直到调用 LSCC 的 Invoke (core/lsccl/lsccl.go 中), args := stub.GetArgs() 获取得到的是第1步所提到的 ccinp, function := string(args[0]) 得到的值是 deploy, 因而之后的 switch-case 会进入 case DEPLOY: 分支。在 case DEPLOY: 分支中, 依次对 args 的每个值进行了检查, 对参数进行了修补, 如 escc/vscc 若为空, 则赋予默认值。最后调用 lsccl.executeDeploy(...), 开始部署 example02。在此罗列一下进入 executeDeploy(...) 的参数: stub 为 ChaincodeStub; chainname 值为 testchain; depSpec 为 CDS, 但是此刻仍是被 Marshal 过的数据; policy 为 args 的第3个参数, 值为 "OR ('Org1MSP.member','Org2MSP.member')"; escc/vscc 由于命令行未指定, 为空值。

8) 在 executeDeploy(...) 中, 需要做如下事情: (a) 先检查 example02 的名字、版本是否可通过 ACL (账户控制列表)。(b) 调用 lsccl.getCCInstance(...), 查看 example02 是否已经存在于链上。(c) 调用 ccpack, err := ccprovider.GetChaincodeFromFS(...) 将 example02 的源码读进一个 CDSPackage 对象 ccpack。然后 cd := ccpack.GetChaincodeData(), 再由 ccpack 生成一个 ChaincodeData 数据对象。(d) 调用 lsccl.getInstantiationPolicy(chainname, ccpack), lsccl.checkInstantiationPolicy(...) 分别获取并检查一个部署策略。(e) 调用 lsccl.createChaincode(stub, cd), 进而直接调用 lsccl.putChaincodeData(stub, cd), 传入 ChaincodeStub 和 ChaincodeData, 执行部署任务。

9) 在 putChaincodeData(stub, cd) 中进行简单地检查之后, 就调用 ChaincodeStub 的函数 stub.PutState(cd.Name, cdbytes), 以 example02 的名字为 key, Marshal 过的 ChaincodeData 数据为 value, 把这一对 key-value 放到账本中。

10) stub.PutState(cd.Name, cdbytes) (core/chaincode/shim/chaincode.go 中定义), 调用了 stub.handler.handlePutState(...) 时触发了 LSCC 的 ShimHandler 的 handlePutState() 函数 (stub 的 handler 是在第7步省略的过

程中，在 `core/chaincode/shim/handler.go` 的 `handleTransaction` 中生成 `ChaincodeStub` 时传入的 `LSCC` 的 `ShimHandler` 实例)。

11) 在 `handlePutState()` 中需要做如下事情：(a) `proto.Marshal(&pb.PutStateInfo{...})`，首先将 `key` 和 `value` 封装，作为一个 `ChaincodeMessage_PUT_STATE` 类型的 `ChaincodeMessage` 消息的 `Payload`，`Txid` 依旧是 `txid`。(b) `handler.sendReceive(msg, respChan)`，调用 `ShimHandler` 将 `ChaincodeMessage_PUT_STATE` 类型的消息异步发送给 `LSCC` 的 `ServerHandler`，然后进入 `select-case` 等待 `ServerHandler` 的回信。注意这里等待的 `respChan`，是 `createChannel` 生成的，这个函数类似于 `ServerHandler` 的 `createTxContext`，都是以 `txid` 为 `key` 在 `map` 中存储通知频道，防止交易重复，且随用随删。

12) `LSCC` 的 `ServerHandler` 收到 `ChaincodeMessage_PUT_STATE` 类型的消息，将只触发状态机的 `enterBusyState` 事件函数。该事件函数整个都是异步执行的。

13) `enterBusyState` 函数一眼看上去很长很麻烦，所以先讲一下函数的布局：(a) 使用 `defer` 作为最后发送消息的地方，发送的消息是 `triggerNextStateMsg`。(b) 如 `ShimHandler` 的 `handleInit` 函数一样，定义了一个 `errHandler` 函数，一旦检查有错误，即将 `triggerNextStateMsg` 赋值后返回，触发 `defer` 发送。若中途没有错误，则顺利到达函数的最后，给 `triggerNextStateMsg` 赋值一个正常的应答消息，然后随着函数的结束触发 `defer` 发送应答。(c) 中部的 `if`，大分支是函数的处理主体，分别处理 `ChaincodeMessage_PUT_STATE`、`ChaincodeMessage_DEL_STATE` 和 `ChaincodeMessage_INVOKE_CHAINCODE` 三类消息，对应执行不同动作，从名字基本就可以判断它各是做什么的：新建一个状态、删一个状态和调用 `chaincode` (改一个状态)，就是熟悉的增删改。接着，函数的具体执行：(a) `handler.createTXIDEntry(msg.Txid)` 是为了防止同一个交易重复执行。(b) `handler.isValidTxSim(msg.Txid...)` 会根据 `txid` 获取 `txContext`，这个交易上下文 `transactionContext` 是在第 7 步省略的过程中，`Excute(...)` 是由最初执行的 `sendExecuteMessage` 中的 `handler.createTxContext(...)` 创建的，创建的同时，也将第 3 步和第 5 步更新到 `ctxt` 中的两个工具取出来后赋值给了 `txContext` 的 `txsimulator` 与 `historyQueryExecutor` 两个字段。这其中交易模拟工具在此之后将用到。(c) `chaincodeID := handler.getCCRootName()` 取出来的是 `ServerHandler` 关于 `LSCC` 的信息 `lsccl:1.0.0`。(d) 进入 `if msg.Type.String() == pb.ChaincodeMessage_PUT_STATE.String()` 分支，`txContext.txsimulator.SetState(...)` 会利用 `txContext` 中的交易模拟工具，最终将 `key` 和 `value`，连同处理 `example02` 的 `lsccl:1.0.0` 一同写入到写集中。写集是一个 `map`，这个 `map` 的线路是：在 `core/ledger/kvledger/txmgmt/rwsetutil/rwset_builder.go` 中的 `RWSetBuilder` 中的 `rwMap` 映射，这个 `map` 以 `lsccl:1.0.0` 为键，映射一个 `nsRWs`，而 `SetState(...)` 最终就是将 `key` 和 `value` 存储在这个 `nsRWs` 中的 `writeMap` (写集) 中。对 `example02` 的部署目前并没有真正提交到账本 (数据库) 中，部署也是一个交易，自然也需要最终提交到账本中，只是目前还没到最终提交的时候。至此，

14) LSCC 的 ShimHandler 收到 ChaincodeMessage_RESPONSE 类型的消息, 触发状态机的 afterResponse 事件函数。该事件函数调用 handler.sendChannel(msg) 向第 11 步提到的 respChan 发送消息, 第 11 步 (b) 的 sendReceive(...) 等待结束。重新定位到 core/chaincode/shim/handler.go 中的 handlePutState, sendReceive(...) 在结束返回后, 将开始一路返回。

16) 继续返回，定位到 `core/chaincode/shim/handler.go` 中的 `handleTransaction` 函数，`handler.cc.Invoke(stub)` 执行完毕。触发 `defer`，向 LSCC 的 `ServerHandler` 发送 `ChaincodeMessage_COMPLETED` 类型的 `ChaincodeMessage` 消息，消息的 `Payload` 是第 15 步所返回的结果。

18) 进入 `if cid.Name == "lsc" && ...` 分支, 对应第4步。cds, err = `putils.GetChaincodeDeploymentSpec(...)` example02 的 CIS 解压出 CDS, `ccid = ccprovider.NewCCContext(...)` 并重新生成一个 `CCContext`, 最后将 example02 的 CDS、`CCContext` 和 `ctxt` 一同传入 `chaincode.Execute(...)`, 第二次进入 Launch-Execute 过程。这次进入没有通过 `core/chaincode/chaincodeexec.go` 中的 `ExecuteChaincode`, 而是直接调用了 `exctransaction.go` 中的 `Execute(...)`。不同于第一次的是: (a) 第一次传入的是 LSCC 的 `CCContext` 和 CIS, 这次传入的是 example02 的 `CCContext` 和 CDS。(b) 第一次时 Launch 的 LSCC 已经 Launch 过, 中途就返回了, 但 example02 没有 Launch 过, 因此这次 Launch 将一直执行下去, 建立 example02 的容器。(c) 第一次传入的是 LSCC 的 CIS, 因此生成的是 `ChaincodeMessage_TRANSACTION` 消息, 第二次传入的是 example02 的 CDS, 因此生成的是 `ChaincodeMessage_INIT` 类型的消息。不同类型的消息都会被传入 `Execute`。

仅供非商业用途或交流学习使用

ExecEnv 值为默认的 ChaincodeDeploymentSpec_DOCKER。据此三点, 可知整个 Launch 中只会进入 if (!chaincodeSupport.userRunsCC || ... 分支, 接着进入 if !(chaincodeSupport.userRunsCC||..., 从文件目录中读取 install 命令放入的 example02 的源码包数据, 进而获取 CDS, 该 CDS 的 CodePackage 包含 example02 的源码数据, 将供后文 example02 的 Docker 容器的建立使用。builder = func() (io.Reader, error) { return platforms.GenerateDockerBuild(cds) } 创建了一个 builder 函数。最后调用 launchAndWaitForRegister(...), 开始对 example 进行 Launch。

20) 在 example02 的 Launch 过程中, 不同于 SCC 的 Launch, builder 在部署中会使用到, example02 启动的是 Docker 容器 DockerVM。一直追溯, 将定位到 core/container/dockercontroller/dockercontroller.go 中的 Start(...) 函数。

21) 在 Start(...) 函数中, 启动了 example02 的 Docker 容器: (a) imageID, err := vm.GetVMName(ccid), 根据 ccid 中保存的三个 ID, 组装一个 example02 的 Docker 镜像 ID, 这个 ID 的规则是小写, 字符范围在只有字母数字、-、.、_ 之内, 否则会用 - 替换, 形式为 %s-%s-%s。(b) client, err := vm.getClientFunc(), 创建一个 go-dockerclient 的客户端对象, 这个对象是实际进行 Docker 容器的基础。(c) containerID := strings.Replace(imageID...), 根据镜像 ID 生成一个容器 ID。(d) attachStdout := viper.GetBool("vm.docker.attachStdout"), 获取一个配置项, 这个配置项默认值是 false, 用于为调试目的而使能 Docker 容器的标准输出和标准错误输出, 这里使用默认值, 即后边的 if attachStdout 分支 (该分支启动了两个 goroutine 分别接收容器的标准输出和标准错误输出) 不会进入。(e) vm.stopInternal(ctxt, ...), 根据 example02 的镜像 ID、容器 ID, 尝试删除可能已经存在的同 ID 的镜像和容器, 为之后的创建扫清障碍。(f) err = vm.createContainer(ctxt, ...), 创建容器, 这个函数主要做的就是首先根据现有数据的指向生成一个 client 认可且可以使用的 Docker 容器配置对象 copts, 这个配置对象除了基本容器的基本信息外, 还指定了一个配置函数, 即 getDockerHostConfig, 然后调用 client.CreateContainer(copts) 创建容器。(g) 进入 if err != nil 分支, 事实上 (f) 将执行失败, 因为创建容器的基础是容器使用的镜像存在, 而当第一次部署的时候, example02 的镜像并不存在, 因此将产生 err == docker.ErrNoSuchImage 的错误, 在这个分支中, 使用了 builder 先创建了 example02 的镜像, 然后重新执行 (f) 创建容器 (h) 执行 prelaunchFunc() 函数, 预 Launch 一下 example02。(i) client.StartContainer(containerID, nil), 启动 example02 的容器, 通过配置对象创建容器的方式可能会在以后根据 Docker 接口的改变而改变。

22) 详解第 21 步创建 example02 的镜像和启动容器的过程。首先概述一下: (a) 使用的是第三方库 github.com/fsouza/go-dockerclient。(b) 创建 ACC 的 Docker 容器并不是简单地使用标准的 docker build+Dockerfile 的机制 (因为这样产生的镜像有体积过大、存在额外安全漏洞、运行笨拙等缺点), 而是先积攒关于 example02 的镜像数据 (Dockerfile 文件、peer

节点的 tls 证书、编译后的可执行程序)，然后创建一个相对轻量级的 ACC 容器（由此可以看出，对 ACC 的容器进行减负，主要是减去要为编译 ACC 而存在的部分，这部分通常使用较少，但占用的空间和资源又相对多）。(c) 编译 example02 源码用到一个容器，这个容器将 core.yaml 中 chaincode.builder 项指定的 fabric-ccenv 作为启动镜像，该镜像由 Fabric 项目提供，在 Getting Started 中下载镜像时会下载，其实应该就是一个能编译 example02 的 Linux 系统容器——ccenv，就是 chaincode environment 的缩写。大致过程为先创建这个容器，然后把 example02 的源码上传到容器中，然后启动容器时执行 go build...，然后再把编译好的可执行程序下载出来。(d) 在 core/chaincode/platforms 下是平台相关的代码，用于生成支持的语言的 ACC 的镜像所需的数据包，platforms.go 是总控文件，car、golang、java 是平台相关的代码，这里只关注 golang 语言。然后详述过程：(a) builder 执行的是 core/chaincode/platforms/platforms.go 中的 GenerateDockerBuild(...)，在这个函数中，先把 example02 容器通过 tls 连接 peer 节点的证书 peer.crt 放入 inputFiles 中，然后调用 generateDockerfile 来生成可用的 Dockerfile 文件（使用 golang 平台的 GenerateDockerfile 来创建了文件头，FROM 命令指定 example02 最终使用 fabric-baseos 镜像，由 core.yaml 中的 chaincode.golang.runtime 项指定，ADD 将编译生成的 example02 的可执行程序压缩包 binpackage.tar 复制并解压到 /usr/local/bin 目录下，还定义了一些 LABEL 和环境变量等），也将它放入了 inputFiles 中。(b) input, output := io.Pipe() 生成了一个管道，连同 go func() {...} 中的 gw、tw 压缩对象，形成了 input<->output<-gw<-tw 的数据流向管道，即向 tw 中写数据，最终会形成压缩包并流向 input。(c) 在新启的 goroutine 中，generateDockerBuild(...) 汇总了 example02 镜像数据。先把证书和 Dockerfile 文件写入 tw，然后调用 golang 的 GenerateDockerBuild(cds, tw)，进而调用 core/chaincode/platforms/util/utlis.go 中的 DockerBuild，依据 fabric-ccenv 镜像创建了一个容器，创建该容器的选项 DockerBuildOptions 指定了三个值：Cmd 指定了编译命令，将 example02 编译成名为 chaincode 的可执行程序并放入 /chaincode/output；InputStream 指定了输入流，该流为 example02 的 CDS.CodePackage；OutputStream 指定了容器的输出流，该流最后也通过调用 cutil.WriteBytesToPackage 写入 tw，随后流向 input。在 DockerBuild 中，所做的就是根据选项先检查 fabric-ccenv 是否存在，若不存在则尝试下载，然后创建、启动 fabric-ccenv 容器，等待编译完成，最后将编译好的 chaincode 从 /chaincode/output/ 中下载到输出流 OutputStream 并删除 fabric-ccenv 容器。(d) 异步执行前一步后直接将 input 返回。返回到第 21 步 (g) 处 builder 执行完毕将 input 返回给 reader，对接上文，reader 就是接收 example 镜像数据。然后通过调用 vm.deployImage，把 reader 作为镜像的输入流（即上下文，可以理解为以此镜像使用 Dockerfile 运行容器时 Dockerfile 能使用的哪个范围下的数据），将 example02 的镜像部署。继续第 21 步的 (g) 向后执行。这里需说明的是，这里启动的容器是 example02 镜像的 Dockerfile 指定的 fabric-baseos，且 ADD 命令会将 binpackage.tar（即名为 chaincode 的 example02 的可执行程序的压缩包）复

制并解压到 /usr/local/bin 目录下，再者 createContainer 创建该容器的时候，配置 Config 中 Cmd 的值（相当于 Dockerfile 中的 CMD）是最初在 core/chaincode/chaincode_support.go 中 getArgsAndEnv(...) 生成的 args = []string{"chaincode", fmt.Sprintf("...)}，所以当 client.StartContainer(containerID, nil) 启动 fabric-baseos 时（准确说是 exmaple02 镜像，fabric-baseos 只是其基础镜像）会执行 example02 的程序 chaincode -peer.address=0.0.0.0:7051。这里要进行清晰地区分，执行 client.StartContainer(containerID, nil) 的是 peer 节点（这个节点可以宿存在主机中，也可以宿存在一个 Docker 容器中），执行 chaincode -peer.address=0.0.0.0:7051 的是 example02 容器。

23) 创建 example02 的两个 Handler。在新运行的 example02 容器中执行 example02 的程序 chaincode，参看源码 examples/chaincode/go/chaincode_example02/chaincode_example02.go，执行的 func main 中直接调用了 shim.Start(new(SimpleChaincode))，该函数在 core/chaincode/shim/chaincode.go 中定义，相当于部署 SCC 时调用的 StartInProc，旨在启动一个 example02 的 ShimHandler 并通过 grpc 主动发送一个 ChaincodeMessage_REGISTER 类型消息给 peer 节点中 example02 的 ServerHandler。传入的 SimpleChaincode 即为 example02 链码对象，相当于 LSCC 的 LifeCycleSysCC。具体的过程如下：(a) SetupChaincodeLogging()，设置 viper 在本容器内获取环境变量值的一些方法，如把前缀设置为 CORE，把 _ 替换为 .，这样 viper.GetString("chaincode.id.name") 就可以获取第 22 步最后启动 example02 容器时设置的 Env 中 CORE_CHAINCODE_ID_NAME=example02:antherversion 的值，其次是获取其他的环境变量以设置日志输出级别等，这些环境变量均是最初在 core/chaincode/chaincode_support.go 中 getArgsAndEnv(...) 生成的，一路被传至 example02 的容器配置中，对照第 22 步中的 createContainer。(b) stream, err := streamGetter(chaincodename)，获取一个 grpc 流，这个流是连接 peer 节点的 ChaincodeSupport 客户端流。其中 peer 的地址是通过 flag.StringVar(&peerAddress, "peer.address"...) 获取的，对应第 22 步最后启动 example02 容器时执行的程序是 chaincode -peer.address=0.0.0.0:7051，即通过 flag 给定了 peer 节点的地址，在此则通过 flag 获取这个地址。这一步执行过后，由于 streamGetter 中执行了 chaincodeSupportClient.Register(...)，因此 peer 节点在 core/chaincode/chaincode_support.go 中的 grpc 服务端的 Register(...) 函数将被调用，进而调用 HandleChaincodeStream。HandleChaincodeStream 新创建了属于 example02 的 ServerHandler 并调用 handler.processStream() 启动了循环接收 ShimHandler 消息的 for 循环。(c) chatWithPeer(chaincodename, stream, cc)，如同 SCC 的部署一样，通过 chatWithPeer，先创建了属于 example02 的 ShimHandler 对象，将 stream 和 cc 赋值给 ShimHandler 相应成员，然后利用 ShimeHandler 向在 peer 节点中的 ServerHandler 发送了一条 ChaincodeMessage_REGISTER 类型消息，最后启动了循环接

收 ServerHandler 消息的进程。(d) (b) 中的 ServerHandler 收到 (c) 中 ShimHandler 发送的 ChaincodeMessage_REGISTER 消息, 开始了注册的过程。这里的注册指的是用 (b) 新建的属于 example02 的 ServerHandler 把第 21 步 (h) 中 prelaunchFunc() 预 Launch 的 Handler 替换掉。直到 example02 的 ServerHandler 和 ShimHandler 均达到 ready 状态。至此, 第二次进行的 Launch-Execute 的 Launch 部分执行完毕, 开始返回。返回定位到对应第 18 步, core/chaincode/exectransaction.go 的 Execute(...) 中, theChaincodeSupport.Launch(...) 执行结束。

24) 继续执行 theChaincodeSupport.Execute(...), 定位到 example02 容器中运行的 ShimHandler 端 core/chaincode/shim/handler.go 的 handleInit(msg) (ShimHandler 接收到 ServerHandler 发来的 ChaincodeMessage_INIT 消息, 状态机触发 beforeInit 事件函数, 进而调用 handleInit(msg)), 在这个函数中: (a) stub := new(ChaincodeStub), stub.init(...) 创建并根据收到的 ChaincodeMessage_INIT 消息初始化了一个 ChaincodeStub。(b) handler.cc.Init(stub), 调用了 ShimHandler 的 cc (即 example02 的 SimpleChaincode 对象) 的 Init 接口, 可以定位到 examples/chaincode/go/chaincode_example02/chaincode_example02.go 中的 Init(stub)。(c) 在 Init(stub) 中, 首先 stub.GetFunctionAndParameters() 获取了 stub 中的 args 中包含的函数和函数所用参数, 即 -c 指定的函数 init, 参数 a, 100, b, 200, 分别看作 a 账户余额 100, b 账户余额 200。然后使用 stub.PutState(A, []byte(strconv.Itoa(Aval))) 和 stub.PutState(B, []byte(strconv.Itoa(Bval))) 将两个账户的初始状态提交。

25) stub.PutState 将触发 ShimHandler 的 handler.handlePutState, 之后的过程类似于第 9~14 步, 只不过这时使用的 ServerHandler 和 ShimHandler 都是 example02 的, 所提交的 key 是 A 的账户名, value 是 A 的余额, 最终也是将这一对 key-value 通过交易模拟工具在 core/chaincode/handler.go 的 enterBusyState 中提交到 example02:anotherversion 的写集 (同第 13 步中的写集) 中。然后返回到 core/chaincode/shim/handler.go 的 handleInit(msg) 中, 随着 handler.cc.Init(stub) 的结束, 触发 defer 发送 ChaincodeMessage_COMPLETED 消息。ServerHandler 收到后通知 core/chaincode/chaincode_support.go 中的 Execute(...) 结束等待并返回, exectransaction.go 中的 Execute(...) 也随之返回。至此, 第二次进行的 Launch-Execute, Execute 部分执行完毕, 开始返回。返回至 core/endorser/endorser.go 的 callChaincode(...), 对应第 18 步。继续返回至 simulateProposal() 中, 进入 if txsim != nil 分支执行 txsim.GetTxSimulationResults(), 获取了交易的读写集 (这里主要是写集中的数据, 是由第 13 步中 LSCC 的写集写入的 example02 的链码数据 ChaincodeData, 此步中 example02 的写集写入的 A/B 两个账户的状态数据, 当前操作的读集里面没有数据), 然后返回至 ProcessProposal(...) 中, 这里罗列一下

`e.simulateProposal(...)` 返回的数据: `cd` 为空; `res` 为 LSCC 部署 `example02` 时的返回结果 `Response`, 成功的结果中包含 `example02` 的 `ChaincodeData`; `simulationResult`, 交易的读写集 (即目前所进行的交易的结果); `ccevent` 为 LSCC 部署 `example02` 时最终返回 `ChaincodeMessage_COMPLETED` 消息时所携带的 `ChaincodeStub` 中定义的事件, 这里为空 (`core/chaincode/shim/handler.go` 的 `handleTransaction` 中)。继续, 由于 `chainID` 不为空, 将继续执行 `e.endorseProposal(...)`, 在此罗列一下传入该函数的参数: `ctxt`; `chainID`, 值为 `testchain`; `txid`, 交易 ID; `signedProp/prop` 对应图中的 `SignedProposal` 和 `Proposal`; `cd/res/res/simulationResult/ccevent` 均为上文返回的数据; `hdrExt.PayloadVisibility` 为空; `hdrExt.ChaincodeId` 只包含一个值为 LSCC 的 `Name` 字段。

26) 在 `endorseProposal(...)` 中, 主要做的就是生成一个供 ESCC 使用的 CIS, 然后通过再次调用 `callChaincode` 来执行背书。具体过程如下: (a) 确定要使用的进行背书的 SCC 的名字 ESCC 和版本号。(b) 根据参数和准备的数据, 生成一个 `eccis`, 这个作用类似于 `ExecuteChaincode` 中的 `createCIS` 和图中的 CIS, `Args` 的值: [0] 函数名, 为空; [1] `Proposal` 的 Header; [2] `Proposal` 的 Payload; [3] []byte 格式的 `ccid`, 这个 `ccid` 的 `Name` 值为 LSCC, `version` 值为 1.0.0; [4] []byte 格式的包含 `example02` 的 `ChaincodeData` 的成功返回结果; [5] 交易读写集数据; [6] 事件, 为空; [7] `payload` 的权限控制, 为空, 当前版本对这个字段并没有使用。后边的背书过程中所用到的数据均来自于此。(c) 调用 `callChaincode`。

27) 在 `callChaincode` 中, 这次只会执行一次 Launch-Execute 过程, 且 Launch 会中途返回, 因为 ESCC 已经被 Launch 过。中途的过程查看 6.11.3 节第 5 步之后, 一直对看到 6.11.6 节的第 6 步, 只不过这期间一直使用的是 ESCC 的两个 Handler, 最后调用到的是 ESCC 的 `Invoke()` 方法对 `example02` 的交易结果进行背书。直接定位到 `core/scc/escc/endorser_onevalidsignature.go` 的 `Invoke`, 传入 `Invoke` 的 `stub` 是在 `core/chaincode/shim/handler.go` 的 `handleTransaction` 中生成的。

28) 在 `Invoke` 中, 只做了两件事: (a) 分别将携带的 `Args` 中的每个值都解压出来, 在此可以与第 26 步 (b) 处的对照, 看看都是哪些数据。(b) 根据解压出来的 `Args` 中的值, 签名并整理应答数据, 最后返回这个数据, 这里不再详述。自此一路返回, 过程省略, 直接定位到 `endorseProposal` 中的 `e.callChaincode(...)` 执行完毕, 对应第 26 步的 (c)。接着返回至 `ProcessProposal()`, 至此, `ProcessProposal()` 全部执行完毕, 返回的数据是 `protos/utills/txutils.go` 中 `CreateProposalResponse` 生成的 `ProposalResponse`, 且该 `ProposalResponse` 的 `Response.Payload` 被赋值为 `example02` 的 `ChaincodeData`。 `ProposalResponse` 的构成: (a) 成员 `Version`, 明确版本固定为 1。(b) 成员 `Endorsement`, 包含了节点的签名者 `Endorser` 和签名者的签名 `Signature`。(c) 成员 `Payload`, []byte 格式的 `ProposalResponsePayload`, 包含 []byte 格式的 `ChaincodeAction` 以及哈希过的 `Proposal`。 `ChaincodeAction` 相当于在描述一个 `chaincode` 动作, 即谁干了什么事儿, 产生了什么后果, 包含的有: `ChaincodeId` 里的数据

是 LSCC 和 1.0.0, Response 数据是 LSCC 部署 example02 时返回的 res, 其中 Payload 包含了 example02 的 ChaincodeData, Results 则包含了 LSCC 部署完 example02 后两条链码的读写集, Events 为空。(4) 一个 Success 的 Response, Payload 为 example02 的 ChaincodeData。这样的一个 ProposalResponse 被返回给 peer/chaincode/instantiate.go 的 instantiate 中, cf.EndorserClient.ProcessProposal(...) 执行完毕。

29) instantiate 继续执行, utils.CreateSignedTx, 结合 Proposal, 返回的 ProposalResponse 生成“一封信”, 即可供 cf.BroadcastClient.Send(env) 使用的 common.Envelope。Envelope 数据的封装过程是一个具体数据到被一层层包装进一个通用数据的过程。

6.12.4 广播

在 peer/chaincode/instantiate.go 的 chaincodeDeploy 中, env, err := instantiate(cmd, cf) 返回的 Envelope 紧接着被 cf.BroadcastClient.Send(env) 进行广播。

peer 命令的广播客户端为在 peer/common/ordererclient.go 中的 broadcastClient, 封装了一个 grpc 连接和一个 AtomicBroadcast_BroadcastClient 客户端对象(这个对象其实是 AtomicBroadcastClient 客户端的一部分, 即 Broadcast 流客户端, 在 protos/orderer/ab.pb.go 中定义)。从所在的文件名就可知, 这是连接 orderer 服务的客户端。ordering 服务客户端的地址是命令行中 -o 指定的, 若未指定, 则是在 peer/common/common.go 中的 GetOrdererEndpointOfChain, 通过向 CSCC 发送请求获取的。这个连接在 instantiate 命令执行之初就已经被建立。

cf.BroadcastClient.Send(env) 将 env 发送到了 orderer/server.go 的 Broadcast(...) 处接收, 自此数据交给了 orderer 服务中进行处理。由于涉及 orderer 服务, 这里只讲大概过程: 即 orderer 服务将 env 数据排序后发送给 peer 节点的 gossip 服务开始散播, 散播后, 最终提交到网络中的每个节点的链(账本)上。

6.12.5 部署后的状态

部署后的状态分为以下几种:

- ❑ example02 的 ChaincodeData 数据被放入交易模拟器的写集中。
- ❑ example02 的镜像被创建, 镜像的上下文包含 Dockerfile 文件, 通过 tls 连接 peer 节点的证书, 可执行程序压缩包。
- ❑ example02 的容器被创建, 单独运行 ShimHandler, 并通过 grpc 与 peer 节点中 ServerHandler 通信。
- ❑ example02 的 ServerHandler 在 peer 节点的 theChaincodeSupport.runningChaincodes.chaincodeMap 中进行了注册。

❑ example02 的 ShimHandler 和 ServerHandler 均处于 ready 状态。

❑ example02 指定的 {"Args":["init","a","100","b","200"]}, 即 A 账户余额 100, B 账户余额 200, 这两个状态被放入交易模拟器的写集中。

example02 部署的数据通过 orderer 服务排序后散播到网络的各个有效节点中并最终提交到各自的链(账本)上(这一点其实不是 peer 部署 example02 本身做的事情, 只是会促成的 orderer 服务和 gossip 服务要做的事情)。

6.13 chaincode 操作步骤

本节是一个在账本上创建资产(键值对)的基本范例。

6.13.1 选择一个代码存放位置

首先需要确保 Go 语言已经被完整安装到系统上。

为 chaincode 应用创建一个位于 \$GOPATH/src/ 目录下的子目录。

使用如下指令:

```
mkdir -p $GOPATH/src/sacc && cd $GOPATH/src/sacc
Now, let's create the source file that we'll fill in with code:
```

使用以下指令创建源文件:

```
touch sacc.go
```

6.13.2 内务处理

首先, 进行内务处理。对于每一个 chaincode, 都会实现预定义的 chaincode 接口, 特别是 Init 和 Invoke 函数接口。所以首先为 chaincode 引入必要的依赖。在此引入 chaincode shim package 和 peer protobuf package。

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)
```

6.13.3 初始化 chaincode

实现 Init 函数。

```
// Init is called during chaincode instantiation to initialize any data.
```

```
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    }
}
```



注意 chaincode 升级同样会调用该函数。当编写的 chaincode 会升级现有 chaincode 时，需要确保适当修正 Init 函数。特别地，如果没有“迁移”操作或其他需要在升级中初始化的东西，则提供一个空的“Init”方法。

调用 ChaincodeStubInterface.GetStringArgs 函数来获取 Init 所需的参数，并进行有效性检查。本例的传入参数是一组键值对。

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }
}
```

将初始状态存入账本。调用 ChaincodeStubInterface 并以键值为参数传入。如果一切正常，会收到表明初始化成功的 peer.Response 返回对象。

```
// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data, so be careful to avoid a scenario where you
// inadvertently clobber your ledger's data!
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}
```


6.13.4 调用 chaincode

添加 Invoke 函数签名：

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The 'set'
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
}
```

类似上述 Init 函数，需要调用 ChaincodeStubInterface 来获取参数。Invoke 函数所需的传入参数正是应用想要调用的 chaincode 的名称。在本例中，只有两个简单的功能函数——set 和 get：前者允许对资产的数值进行设定，后者允许获取当前资产的状态。先调用 ChaincodeStubInterface.GetFunctionAndParameters 来获取 chaincode 应用所需的函数名与参数。

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

}
```

使 set 与 get 这两个函数名正式生效，并调用这些 chaincode 应用函数，经由 shim.Success 或 shim.Error 函数返回一个合理的响应。这两个 shim 成员函数可以将响应序列化为 gRPC protobuf 消息。

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
        result, err = set(stub, args)
    } else {
        result, err = get(stub, args)
    }
    if err != nil {
        return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}
```

6.13.5 实现 chaincode 应用

此 chaincode 应用实现了两个函数，并可以被 Invoke 函数调用，下面将实现这些函数。注意，就像上文一样，通过调用 chaincode shim API 中的 ChaincodeStubInterface.PutState 和 ChaincodeStubInterface.GetState 函数来访问账本。

```
// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

6.13.6 整合全部代码

编写 main 函数，它将调用 shim.Start 函数。下面是包含整个 chaincode 程序的代码：

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct {
```

```

}

// Init is called during chaincode instantiation to initialize any
// data. Note that chaincode upgrade also calls this function to reset
// or to migrate data.
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
        return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // Set up any variables or assets here by calling stub.PutState()

    // We store the key and the value on the ledger
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // Extract the function and args from the transaction proposal
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
        result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
        result, err = get(stub, args)
    }
    if err != nil {
        return shim.Error(err.Error())
    }

    // Return the result as success payload
    return shim.Success([]byte(result))
}

// Set stores the asset (both key and value) on the ledger. If the key exists,
// it will override the value with the new one
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))

```



```

    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// Get returns the value of the specified asset key
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}

// main function starts up the chaincode in the container during instantiate
func main() {
    if err := shim.Start(new(SimpleAsset)); err != nil {
        fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
    }
}

```

6.13.7 编译 chaincode

编译 chaincode 的代码如下：

```

go get -u --tags nopkcs11
github.com/hyperledger/fabric/core/chaincode/shim
go build --tags nopkcs11

```

成功则可以进行下一步：测试 chaincode。

6.13.8 在开发者模式下测试

通常，chaincode 由 peer 节点启动并维护。不过，在“开发者模式”下，chaincode 可以由用户创建并启动。当用户处于以快速编码、构建、运行、调试的循环周期为主的 chaincode 开发阶段时，该模式十分有用。借助构建自带区块链样例网络时已经预先生成好的 order 和 channel 来启动“开发者模式”。这样，用户可以立即编译 chaincode 并调用函数。

6.13.9 安装 Hyperledger Fabric 样例


如果你之前还没有进行过这一步，请先安装 Hyperledger Fabric Sample。下面进入到安

装好的 fabric-samples 下的 chaincode-docker-devmode 目录。


```
cd chaincode-docker-devmode
```

6.13.10 下载 Docker 镜像

根据下载样例中自带的 Docker 构建脚本，我们需要四个 Docker 镜像来确保“开发者模式”成功运行。如果你已经安装了 fabric-samples 克隆仓库，并按照指示下载了 platform-specific-binaries，那么你的本地理应早已安装好了所需的 Docker 镜像。

 **注意** 如果你选择手动拉取镜像，那么务必将其重新标记为 latest。输入 docker images 指令可以方便地查询本地的 Docker 镜像列表。你应该会看到类似下面的内容：

docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hyperledger/fabric-tools	latest	e09f38f8928d	4 hours ago	1.32 GB
hyperledger/fabric-tools	x86_64-1.0.0	e09f38f8928d	4 hours ago	1.32 GB
hyperledger/fabric-orderer	latest	0df93ba35a25	4 hours ago	179 MB
hyperledger/fabric-orderer	x86_64-1.0.0	0df93ba35a25	4 hours ago	179 MB
hyperledger/fabric-peer	latest	533aec3f5a01	4 hours ago	182 MB
hyperledger/fabric-peer	x86_64-1.0.0	533aec3f5a01	4 hours ago	182 MB
hyperledger/fabric-ccenv	latest	4b70698a71d3	4 hours ago	1.29 GB
hyperledger/fabric-ccenv	x86_64-1.0.0	4b70698a71d3	4 hours ago	1.29 GB

 **注意** 如果你通过 download-platform-specific-binaries 来获取镜像，那么将会看到更多的镜像资源列表。不过这里我们只关心以上四个。现在请在 chaincode-docker-devmode 目录下打开三个独立的终端。

6.13.11 1号终端

执行如下命令：

```
docker-compose -f docker-compose-simple.yaml up
```

上述指令启动了一个带有 SingleSampleMSPSoloorderer profile 的网络，并将节点在“开发者模式”下启动。它还启动了另外两个容器：一个包含 chaincode 运行环境；另一个是 CLI 命令行，可与 chaincode 进行交互。创建并加入 channel（管道）的指令内嵌于 CLI 容器中，所以我们下面马上跳转到 chaincode 调用部分。

6.13.12 2号终端

执行如下命令：

```
docker exec -it chaincode bash
```

执行完上述指令后，你应该会看到如下内容：

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

（此时已经进入 chaincode 容器）下面编译你的 chaincode：

```
cd sacc
go build
```

现在运行 chaincode：

```
CORE_PEER_ADDRESS=peer:7051 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

chaincode 被 peer 节点启动，chaincode 日志表明 peer 节点成功注册。



注意 现阶段 chaincode 还没有与任何 channel 关联。这会在接下来使用 instantiate 指令后实现。

6.13.13 3 号终端

即便处于 --peer-chaincodedev 模式，安装 chaincode 这一步仍必不可少，这样生命周期系统 chaincode 才能正常进行检查。也许这一步会在日后的 --peer-chaincodedev 模式中省去。下面进入 CLI 容器进行 chaincode 调用。

```
docker exec -it cli bash
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

现在执行一次将 a 的值设为 20 的调用：

```
peer chaincode invoke -n mycc -c '{"Args":["set","a","20"]}' -C myc
```

最后查询 a 的值，会看到 20。

```
peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
```

6.13.14 测试新的 chaincode

虽然只实现了 sacc，不过可以通过将不同的 chaincode 添加到 chaincode 子目录下重启网络来轻松地测试它们。重启它们将可在 chaincode 容器中被访问。

MSP 成员服务提供者

Fabric 提供一个所有参与者都有一个明确身份的交易网络，并以此网络为基础。公钥基础设施（PKI, Public Key Infrastructure）被用来生成加密证书（cryptographic certificates），这些被试用于组织、网络的构成，终端用户或客户端的应用。结果，数据权限控制可以在广域网络（broader network）和层级频道（channel level）被操作和支配。这个 Fabric 的“授权”的概念，与 channel 的存在与能力相配对，帮助实现了隐私和机密为主要考虑的地址方案（address scenarios）。

本章将让你更好地 Membership Service Providers（MSP）理解加密操作、签名、校验、认证（cryptographic implementations, and the sign, verify, authenticate）等在 Fabric 中的应用。

7.1 MSP 的设计思路

本章着力于提供关于 MSP 的安装上最好的实践细节。

MSP 是一个着眼于提供一个会员关系操作构架的抽象，作为系统的一个组成部分。

特别的，MSP 抽象了所有加密机制和协议，隐藏了存在的争论（issuing）、验证证书和用户认证。一个 MSP 可能定义它自己的一套身份标识、验证规则，以及签名的生成和认证方法。

Fabric 区域链网络可能会被一个或多个 MSP 管理，他们提供了模块化的会员操作和在不同的会员关系标准和构架间协同工作的能力。

下文我们将详细介绍 MSP 的操作体系，并讨论关于 MSP 方面的实战练习。

7.1.1 MSP 配置

为了启动一个 MSP，它的配置文件必须在每个 peer 和 orderer(去使能 peer 并排序签名)的本地中被指定，开启频道上的 peer、orderer、客户端的身份认证，以及所有会员各自的签名认证。

首先，MSP 需要指定一个名字在网络中代表自己，如 msp1,org2 等。这个名字在其会员关系规则中代表一个 channel 中的一个共同体、组织或者组织分部。也被引用作为 MSP 对象的 ID (MSP Identifier)。每个 MSP 的 ID 是唯一的。例如，若检测出来存在两个相同 ID 的 MSP 实例存在于系统初创的 channel，orderer 将会启动失败。

MSP 在默认操作的情况下，一个参数集合需要被指定 (也就是一个 MSP 需要哪些成员)，用于身份认证 (identity/certificate validation) 和签名认证 (signature verification)。这些参数由 RFC5280 指定，包括：

- ❑ 一个自己签名出的证书 (self-signed) 列表，用于作为信任机制的根证书列表 (root of trust，这个根证书应该是：把自己签出来的证书分发给他人，他人拿着证书来核对时，在根证书列表中查询核对，如果证书存在于这个列表中，则说明这个人受信任的)。该列表可以用 root CAs 表示。
 - ❑ 考虑到证书验证，提供一个用于代表 (作为) 中间人 CAs 证书的 X.509 证书列表。这些 CA 证书应该是在 root of trust 中被鉴定过的。中间人 CA 证书是可选的 (参数)。该列表可以用 intermediate CAs 表示。
 - ❑ 一个 X.509 证书列表，该列表中的证书包含可验证的证书路径，而这些路径对应指向 root of trust 中的证书。所以该证书列表代表着 MSP 管理员的角色。这些证书的拥有者被允许发起请求改变该 MSP 的配置 (即改变 root CAs、中间人 CAs)。该列表可以用 administrator CAs 表示。
 - ❑ 一个组织单位列表，MSP 有效的成员应该包含在这些组织单位的 X.509 证书中。这是一个可选配置参数，当多个组织使用同一个 root CAs，intermediate CAs 并为组织成员保留了一个 OU 字段时，才会使用这个列表。该列表可以用 OU List 表示。
 - ❑ 一个废除证书列表，每一个证书都可以对应到 root CAs 或 intermediate CAs (废除的证书也是从这里来的，而不是凭空产生的)，这是一个可选参数。该列表可以用 CRLs 表示。
 - ❑ 一个用于 TLS 的 X.509 根证书列表 (self-signed (X.509) certificates, TLS root of trust)。
 - ❑ 一个用于 TLS 的 X.509 中间人证书列表。这是一个可选参数。
- 对于一个 MSP 对象实例来说，有效的身份需要满足以下条件：
- ❑ X.509 格式的证书，并包含可验证的证书所在路径，该路径对应到 root CAs。
 - ❑ 不包括在任何一个 CRL 中。
 - ❑ 在 X.509 格式的证书结构体中的 OU 字段里，存在一个或多个属于 OU List 的组织单位。

更多的 MSP 身份的有效性的信息，请阅读 MSP Identity Validity Rules。

除了验证相关参数，对于 MSP 来说，使节点能够去签名或授权，你需要指定：

用于节点签名的签名匙 (signing key)，当前只支持 ECDSA keys。节点的 X.509 证书，该证书必须是在 MSP 的一些列参数中 (root CAs 和中间人 CAs 中) 的有效证书。需要重点注意的是，MSP 中的身份信息永不消失，只能被加入到 CRLs 中。此外，当前不支持强制废除 TLS 的证书。

7.1.2 如何生成 MSP 证书和它们的签名匙

为了生成 X.509 证书去填 MSP 的配置，我们可以使用 openssl。这里强调一下，Fabric 不支持证书中包含 RSA keys。

作为替代，我们可以使用 cryptogen tool，在 Getting Started 中有详述。

Hyperledger Fabric CA 这个项目也可以用来生成 MSP 配置所需的 keys 和证书。

7.1.3 MSP setup on the peer & orderer side

为了建立本地 MSP (local MSP)，无论是 peer 还是 orderer 上的，管理者都要创建一个目录，如 \$MY_PATH/mspconfig，包含如下子目录和文件：

1. admincerts 目录，包含 pem 文件，每个 pem 文件对应一个 administrator 证书，即 administrator CAs。
2. cacerts 目录，包含每个的 pem 对应一个 root 证书，即 root CAs。
3. (可选) intermediatecerts 目录，对应中间人 CA。
4. (可选) config.yaml 文件，包含关于组织单位 OU 的信息。
5. (可选) crls 目录，包含相关的 CRLs。
6. keystore 目录，包含一个含有节点签署密匙的 PEM 文件；我们着重于当前不被支持的 RSA 秘钥。
7. signcerts 目录，包含一个含有 X.509 整数的 PEM 文件。
8. (可选) tlscacerts 目录，包含与 TLS 根 CA 证书一一对应的 PEM 文件。
9. (可选) tlsintermediatcerts 目录，包含一些与中间 TLS CA's 证书节点一一对应的 PEM 文件。

7.1.4 Channel MSP setup

在系统初创的时候，出现在网络中所有 MSP 的验证元素 (即各种证书、配置) 都需要被指定过 (即必须已经存在)，并被包含到系统 channel 的创世纪块 (genesis block) 中。回忆一下，MSP 验证元素由 MSP 身份标识 (MSP identifier)、root CAs、intermediate CAs、admin CAs、OU List、CRLs 组成。在 orderer 进行体系解析时，系统的创世纪块被用于 orderer，

据此 orderer 被允许去认证 channel 的创建请求（等于说 orderer 有认证的材料了，如果创建 channel 的请求合法才通过该请求）。如果创世纪块包含了两个相同身份标识的 MSP，orderer 将拒绝此块，自然地，整个网络的引导建立也会失败。

对于应用（application）的 channel，其验证的各个组成部分，也就是（只能是）管理该 channel 的 MSP，必须存在于 channel 的创世纪块中。我们强调，在 channel 创建一个或多个 peer 加入该 channel 之前，保证包含在该 channel 的创世纪块（或者是最新的配置块）中的 MSP 配置信息的正确性是应用的责任。

当在 configtxgen 工具（configtxgen tool）帮助下引导启动一个 channel 时，通过将 MSP 的验证元素（如 root CAs、中间人 CAs 等）放置到 MSP 专用的配置目录，并在 configtx.yaml 文件中设置相应的选项，我们可以使用 configtxgen 工具配置该 channel 的 MSP。

重新配置一个 channel 上的 MSP，包括该 MSP 相关的废止证书列表（CRLs）的公告，通过该 MSP 中的 Admin CAs 中的一员（即拥有 Admin CA 证书的节点，也叫作 admin 节点）创建一个配置更新对象（config_update object），可以实现一个 channel 上 MSP 的重新配置。如此之后，被 admin 管理的客户端应用将宣布这个针对该 MSP 所在的 channel 的更新。

7.1.5 最佳实践

本节我们将详述 MSP 配置在一般情况下的最好的实践（best practices）。

1. 组织单位和 MSP 之间的映射

我们要求 MSP 与组织（organizations）之间是一一对一的映射。如果选择一个不同的映射类型，需要考虑如下内容：

一个组织雇佣多种（个）MSP。这对应了一个组织包含多个不同部门，出于独立管理或隐私（权限）的原因，每个部门有一个 MSP 代表的情况。在这种情况下，一个 peer 只能被一个 MSP 所拥有（代表一个层级上的一个部门），也不能从同一组织中其他的 MSP 中识别出其他 peer 的身份。这样做的意义在于，一个节点可能通过 gossip 只与从属于该节点的部门（子部门）分享组织范围内的数据，而不是与组织内的所有成员分享。

多个组织使用一个 MSP。这对应了多个组织以类似的关系被管理组成一个协作整体（consortium，如多个学校组织联合起来成立一个学校联盟）的情况。这里我们需要知道的是，一个组织中的多个 peer 将传递组织范围内的消息给同属与同一个 MSP 下的 peer，而且会忽略这些 peer 是否是属于自己的同一个组织。这是一个 MSP 定义的颗粒度（granularity）和 peer 节点配置的局限。一个组织有不同的部门（称为组织单位 -organizational units）。其中的任意一个部门可以给予多个不同 channel 访问权限，如图 7-1 所示。

两种方法如下。

第一种方法：定义一个 MSP，将所有组织的成员纳入到该 MSP 的成员关系（membership）中（即将所有成员的证书之类的数据纳入到该 MSP）。该 MSP 的配置将由 root CAs、

intermediate CAs 和 admin CA 组成，会员身份将包括所有的 OU。策略 (Policies) 被定义为获取指定 OU 的所有成员，该策略可以用于组成一个 channel 的读写 (read/write) 策略，或者一个 chaincode 的背书 (endorsement) 策略。该方法的一个限制就是 gossip peer 会把那些在其本地 MSP 中的 peer (即 gossip 节点的本地 MSP 中的所有成员) 当作是同一个组织内的成员，结果也自然会向这些节点传播组织范围内的数据 (比如它们的状态)。

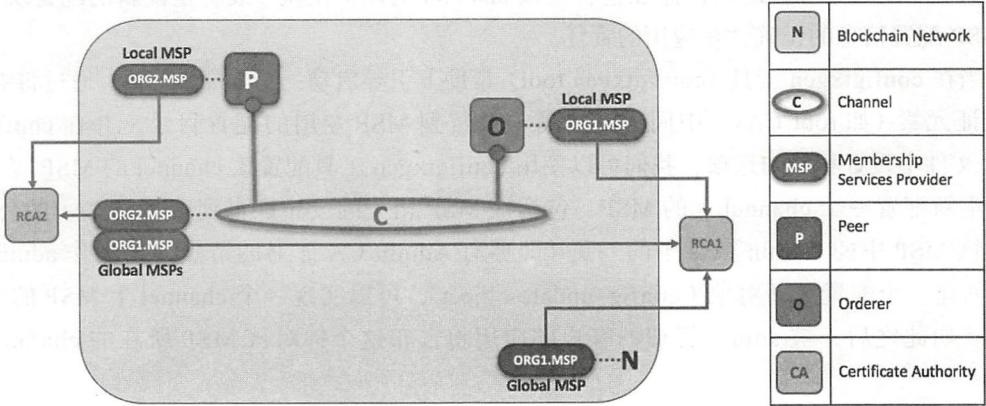


图 7-1 MSP 组织关系图

第二种方法：为每一个部门定义一个 MSP，这将涉及每一个部门的指定问题，即每一个部门都需要指定 root CAs、intermediate CAs 和 admin Certs，这样的话，在各个 MSP 之间没有重叠的证书路径。这就是说，比如，不同的 intermediate CA 对应不同的子部门。这里的缺点是要管理多个 MSP，但这规避了上一种方法所出现的问题。我们也可以通过使用 MSP 的 OU 扩展为每一个部门定义一个 MSP。

从同一个组织中的众多 peer 中分离客户端

在很多情况下，从身份自身获取身份的类型是很必要的 (如背书被担保是由 peer 这种类型的节点产生，而不是客户端或扮演 orderer 的节点)。

以下是对该需求的有限支持。

一种实现这种分离的方法是为每一类节点都创建一个单独的 intermediate CA——一个客户端的，一个 peer/orderer 的；配置两个不同的 MSP——一个客户端的，一个 peer/orderer 的。该组织应该 (可以) 访问的频道，需要同时包含这两个 MSP，而背书策略 (定义) 只使用代表 peer 节点的 MSP。最终，这个组织将会被映射到两个 MSP 实例中，peer 自然可以与 client 相互作用 (交流) 了。

gossip 不会被大幅影响，因为同一个组织的所有 peer 依然属于同一个 MSP。peer 的节点可以限制某些系统 chaincode 运行到本地的基于策略的 MSP。例如，如果该请求被客户端类型的 MSP (终点用户位于请求的起源处) 的 admin 签名的话，peer 将只会执行 “joinChannel” 请求。我们可以绕开这点的矛盾，如果我们接受，对于 peer/orderer 的

MSP 来说，客户端能成为该 MSP 成员的只能是该 MSP 的 admin。

此种方法需要考虑的另一点是，基于请求源头在它本地 MSP 中所处的地位，peer 授权事件注册请求。很明显，因为请求源头是客户端，相较于被请求的 peer，请求源头总是注定属于不同的 MSP，并且被请求的 peer 将拒绝该请求。

2. Admin 与其证书

MSP 的 admin 证书是独一无二的，与该 MSP 中 root CAs 或者 intermediate CAs 中的证书都不同。这是一种通用的实践：将管理成员关系的责任从证书的验证讨论中脱离出来。

3. intermediate CA 黑名单

正如前文提到的，重新配置一个 MSP 通过重新配置机制（对本地 MSP 实例手工重配，或者通过构建一个 channel 中 MSP 实例的一个配置升级消息）实现。很明显，这里有两种方法去确保一个 intermediate CA 被拉入黑名单：

第一种是重新配置一个不再包含黑名单证书的 MSP。对于本地配置，这意味着从 intermediatecert CAs 目录中移除黑名单证书。第二种是重新配置 MSP 的 CRL，将黑名单证书从 root CAs 中加入到 CRL 中（也就是拉黑）。当前的 MSP 操作，我们仅支持第一种方法，这种方法更简单，也不需要拉黑不信任的证书。

4. CAs 和 TLS CAs

MSP 的 root CAs 与 TLS 的 root CAs（相对于中间人 CA 来说）需要定义在不同的目录中。这样是为了避免在两种不同类型的证书之间产生混淆。关于这点来说，不禁止，但是最好在产品中避免。

7.2 MSP 实现剖析

MSP 是 Fabric 中的成员服务提供者，为 Fabric 的各个基础组件提供一系列的身份认证服务。

7.2.1 目录结构

- ❑ msp

- ❑ common/localmsp

成员服务提供者（MSP）是一个提供抽象化成员操作框架的组件。MSP 将颁发与校验证书，以及用户认证背后的所有密码学机制和协议都抽象了出来。一个 MSP 可以自己定义身份，以及身份的管理（身份验证）与认证（生成与验证签名）规则。

一个 Hyperledger Fabric 区块链网络可以被一个或多个 MSP 管理。提供了模块化的成员操作，以及兼容不同成员标准与架构的互操作性。

7.2.2 MSP 配置

在 MSP 初始化时 channel 上的所有节点 (peer、order) 都需要指定其配置, 并在 channel 上启用 peer 节点、order 节点及 client 的身份验证与各自的签名验证。每个 MSP 都必须有独一无二的 ID 进行标识, 当 MSP 的成员管理规则表示一个团体, 组织或组织分工时, 该名称会被引用。当具有相同标识符的两个 MSP 实例在系统 channel 原始块中被检测到时, 那么 orderer 节点的启动将以失败告终。

在 MSP 的默认情况下, 身份 (证书) 验证与签名验证需要指定一组参数。这些参数推导自 RFC5280, 参数包括:

- ❑ 一个自签名的证书列表 (满足 X.509 标准) 以构成信任源。
- ❑ 一个用于表示该 MSP 验证过的中间 CA 的 X.509 的证书列表, 用于证书的校验。这些证书应该被信任源的一个证书所认证, 中间 CA 是可选参数。
- ❑ 一个具有可验证路径的 X.509 证书列表 (该路径通往信任源的一个证书), 以表示该 MSP 的管理员。这些证书的所有者对 MSP 配置的更改要求都是经过授权的 (例如根 CA 和中间 CA)。
- ❑ 一个组织单元列表, 该 MSP 的合法成员应该将其包含进它们的 X.509 证书。这是一个可选的配置参数, 举个例子: 当多个组织使用相同信任源、中间 CA 以及组织为它们的成员保留了一个 OU 区的时候, 会配置此参数。
- ❑ 一个证书吊销列表 (CRLs) 的清单, 清单的每一项对应于一个已登记的中间或根 MSP 证书颁发机构 (CA), 这是一个可选的参数。
- ❑ 一个自签名的证书列表 (满足 X.509 标准) 以构成 TLS 信任源, 服务于 TLS 证书。
- ❑ 一个表示该 provider 关注的中间 TLS CA 的 X.509 证书列表。这些证书应该被 TLS 信任源的一个证书所认证; 中间 CA 则是可选参数。
- ❑ 对于该 MSP 实例, 有效的身份应符合以下条件:
- ❑ 它们应符合 X.509 证书标准, 且具有一条可验证的路径 (该路径通往信任源的一个证书)。
- ❑ 它们没有包含在任何 CRL 中。
- ❑ 它们列出了一个或多个 MSP 配置的组织单元 (列出的位置是在它们 X.509 证书结构的 OU 区内)。

除了验证相关参数外, 为了使 MSP 可以对已实例化的节点进行签名或认证, 需要指定:

- ❑ 用于节点签名的签名密钥 (目前只支持 ECDSA 密钥)。
- ❑ 节点的 X.509 证书, 对 MSP 验证参数机制而言是一个有效的身份。



注意 MSP 身份永远不会过期, 它们只能通过添加到合适的 CRL 上来被撤销。此外, 现阶段不支持吊销 TLS 证书。

1. cert.go

cert.go 用于实现 certificate 相关结构体。

(1) struct

包含一个 certificate 结构体：

```
type certificate struct {
    Raw          asn1.RawContent //
    TBSCertificate tbsCertificate
    SignatureAlgorithm pkix.AlgorithmIdentifier
    SignatureValue  asn1.BitString
}
```

(2) function

判断是否是 ECDSA 算法签名的证书：

```
func isECDSASignedCert(cert *x509.Certificate) bool
```

判断证书的签名是否对应父证书共钥，不对应则重新生成签名：

```
func sanitizeECDSASignedCert(cert *x509.Certificate, parentCert *x509.Certificate)
(*x509.Certificate, error)
```

解析出一个 certificate：

```
func certFromX509Cert(cert *x509.Certificate) (certificate, error)
```

将 certificate 转换成 PEM 格式：

```
func (c certificate) String() string
```

将 x509.Certificate 格式的证书转换成 PEM 格式：

```
func certToPEM(certificate *x509.Certificate) string
```

2. configbuilder.go

用于初始化 BCCSP，以及读取 MSP 文件夹下的各种证书信息，初始化 MSPConfig 信息。

(1) struct

msp 文件夹下的文件：

```
const (
    cacerts          = "cacerts"
    admincerts       = "admincerts"
    signcerts        = "signcerts"
    keystore          = "keystore"
    intermediatecerts = "intermediatecerts"
    crlsfolder        = "crls"
    configfilename    = "config.yaml"
    tlscacerts        = "tlscacerts"
    tlsintermediatecerts = "tlsintermediatecerts"
)
```

用于对各个证书中的 ou 进行校验:

```
type OrganizationalUnitIdentifiersConfiguration struct {
    Certificate          string `yaml:"Certificate,omitempty"`
    OrganizationalUnitIdentifier string `yaml:"OrganizationalUnitIdentifier,omitempty"`
}
type Configuration struct {
    OrganizationalUnitIdentifiers []*OrganizationalUnitIdentifiersConfiguration `yaml:"OrganizationalUnitIdentifiers,omitempty"`
}
```

(2) function

读文件:

```
func readFile(file string) ([]byte, error)
```

读取 Pem 文件:

```
func readPemFile(file string) ([]byte, error)
```

获取文件夹下的一系列证书:

```
func getPemMaterialFromDir(dir string) ([] []byte, error)
```

设置 BCCSP 配置:

```
func SetupBCCSPKeystoreConfig(bccspConfig *factory.FactoryOpts, keystoreDir string) *factory.FactoryOpts
```

获取 LocalMsp 配置:

```
func GetLocalMspConfig(dir string, bccspConfig *factory.FactoryOpts, ID string) (*msp.MSPConfig, error)
```

获取校验 MSP 配置:

```
func GetVerifyingMspConfig(dir string, ID string) (*msp.MSPConfig, error)
```

获取 MSP 配置:

```
func getMspConfig(dir string, ID string, sigid *msp.SigningIdentityInfo) (*msp.MSPConfig, error)
```

获取分层的 MSP 配置:

```
func GetIdemixMspConfig(dir string) (*msp.MSPConfig, error)
```

3. idmixmsp.go

身份混合器 MSP。在某些使用情况下,使用加密协议来保护用户的隐私权非常重要,因为这些协议在签名、认证和传输认证属性的过程中都是如此。在 Fabric 中,这是通过 Identity Mixer 实现的,该混合器具有类似于标准 X.509 证书所保证的信任模型和安全性保

证，但其底层加密算法可有效提供高级隐私功能，如“不可链接性”和最小属性披露。

4. identities.go

包括了 `signingidentity` 实体，代表了一个 `msp` 身份实体，其中包括一系列证书以及标识符等，还包括了创建 `signingidentity` 的实例化方法以及签名方法等。

(1) struct

定义 `identity` 实体所包含的属性：

```
type identity struct {
    // id contains the identifier (MSPID and identity identifier) for this instance
    id *IdentityIdentifier

    // cert contains the x.509 certificate that signs the public key of this instance
    cert *x509.Certificate

    // this is the public key of this instance
    pk bccsp.Key

    // reference to the MSP that "owns" this identity
    msp *bccspmsp
}

type signingidentity struct {
    // we embed everything from a base identity
    identity

    // signer corresponds to the object that can produce signatures from this identity
    signer crypto.Signer
}
```

(2) function

签名如下：

```
func (id *signingidentity) Sign(msg []byte) ([]byte, error)
```

5. msp.go

定义了一系列 MSP 相关的接口：`MSPManager`、`MSP`、`Identity`、`SigningIdentity`。

```
type MSPManager interface {

    // IdentityDeserializer interface needs to be implemented by MSPManager
    IdentityDeserializer

    // Setup the MSP manager instance according to configuration information
    Setup(msps []MSP) error

    // GetMSPs Provides a list of Membership Service providers
    GetMSPs() (map[string]MSP, error)
}
```

```

type MSP interface {
    // IdentityDeserializer interface needs to be implemented by MSP
    IdentityDeserializer

    // Setup the MSP instance according to configuration information
    Setup(config *msp.MSPConfig) error
    // GetType returns the provider type
    GetType() ProviderType

    // GetIdentifier returns the provider identifier
    GetIdentifier() (string, error)

    // GetSigningIdentity returns a signing identity corresponding to the provided
    // identifier
    GetSigningIdentity(identifier *IdentityIdentifier) (SigningIdentity, error)

    // GetDefaultSigningIdentity returns the default signing identity
    GetDefaultSigningIdentity() (SigningIdentity, error)

    // GetTLSRootCerts returns the TLS root certificates for this MSP
    GetTLSRootCerts() [][]byte

    // GetTLSIntermediateCerts returns the TLS intermediate root certificates for
    // this MSP
    GetTLSIntermediateCerts() [][]byte

    // Validate checks whether the supplied identity is valid
    Validate(id Identity) error

    // SatisfiesPrincipal checks whether the identity matches
    // the description supplied in MSPPrincipal. The check may
    // involve a byte-by-byte comparison (if the principal is
    // a serialized identity) or may require MSP validation
    SatisfiesPrincipal(id Identity, principal *msp.MSPPrincipal) error
}

type Identity interface {
    // ExpiresAt returns the time at which the Identity expires.
    // If the returned time is the zero value, it implies
    // the Identity does not expire, or that its expiration
    // time is unknown
    ExpiresAt() time.Time

    // GetIdentifier returns the identifier of that identity
    GetIdentifier() *IdentityIdentifier

    // GetMSPIdentifier returns the MSP Id for this instance
    GetMSPIdentifier() string
}

```

```

// Validate uses the rules that govern this identity to validate it.
// E.g., if it is a fabric TCert implemented as identity, validate
// will check the TCert signature against the assumed root certificate
// authority.
Validate() error

// GetOrganizationalUnits returns zero or more organization units or
// divisions this identity is related to as long as this is public
// information. Certain MSP implementations may use attributes
// that are publicly associated to this identity, or the identifier of
// the root certificate authority that has provided signatures on this
// certificate.
// Examples:
// - if the identity is an x.509 certificate, this function returns one
//   or more string which is encoded in the Subject's Distinguished Name
//   of the type OU
// TODO: For X.509 based identities, check if we need a dedicated type
//       for OU where the Certificate OU is properly namespaced by the
//       signer's identity
GetOrganizationalUnits() []*OUIdentifier

// Verify a signature over some message using this identity as reference
Verify(msg []byte, sig []byte) error

// Serialize converts an identity to bytes
Serialize() ([]byte, error)

// SatisfiesPrincipal checks whether this instance matches
// the description supplied in MSPPrincipal. The check may
// involve a byte-by-byte comparison (if the principal is
// a serialized identity) or may require MSP validation
SatisfiesPrincipal(principal *msp.MSPPrincipal) error
}

type SigningIdentity interface {

    // Extends Identity
    Identity

    // Sign the message
    Sign(msg []byte) ([]byte, error)

    // GetPublicVersion returns the public parts of this identity
    GetPublicVersion() Identity
}

```

6. mspimpl.go

包含 `bccspmsp` 结构体，实例化方法如下：

```
func NewBccspMsp() (MSP, error)
```


上面的代码是在 `msp.go` 中的 MSP 接口的具体实现。

7. mspmgrimpl.go

在 `msp.go` 中的 MSPManager 接口的具体实现如下代码所示，是 MSP 实例的一个管理者。

```
type mspManagerImpl struct {
    // map that contains all MSPs that we have setup or otherwise added
    mspMap map[string]MSP

    // error that might have occurred at startup
    up bool
}
```

8. msp/cache/cache.go

使用了 `github.com/golang/groupcache/lru` 中的 lru cache 缓存，以此来提高性能。

9. msp/mgmt/deserializer.go

对 local 和 channel deserializers 的支持接口如下：

```
type DeserializersManager interface {
    Deserialize(raw []byte) (*mspproto.SerializedIdentity, error)

    // GetLocalMSPIdentifier returns the local MSP identifier
    GetLocalMSPIdentifier() string

    // GetLocalDeserializer returns the local identity deserializer
    GetLocalDeserializer() msp.IdentityDeserializer

    // GetChannelDeserializers returns a map of the channel deserializers
    GetChannelDeserializers() map[string]msp.IdentityDeserializer
}
```

10. msp/mgmt/mgmt.go

从指定文件夹获取 LocalMsp，并进行初始化：

```
func LoadLocalMsp(dir string, bccspConfig *factory.FactoryOpts, mspID string) error
```

获取 LocalMsp 实例：

```
func GetLocalMSP() msp.MSP
```

`msp/mgmt/mgmt.go` 主要是对 `msp` 文件夹下的其他方法的封装。

11. msp/mgmt/principal.go

`principal.go` 用于 MSP 实例获取。

根据不同的角色获取相应的实例。如果角色不存在，则返回零值，并返回相应的 error 信息。

```
func (m *localMSPPrincipalGetter) Get(role string) (*msp.MSPPrincipal, error)
```

12. localmsp/signer.go

用于签名, 构造 SignatureHeader, 包括签名的实体 creator 和随机数 nonce:

```
func (s *mspSigner) NewSignatureHeader() (*cb.SignatureHeader, error)
```

签名消息:

```
func (s *mspSigner) Sign(message []byte) ([]byte, error)
```

Gossip 节点间的流言蜚语

Gossip 协议是分布式系统用于保证分布式一致性的一种方式，也是在 Fabric 中传递各种信息（包括区块信息）的手段。

8.1 Gossip 协议原理解析

HyperLedger Fabric 通过把工作节点分解为执行交易（背书和提交）节点和交易排序节点来优化区块链网络性能、安全性和可扩展性。这种解耦网络操作的方式需要一个安全、可靠、可扩展的数据分发协议来保证数据的完整性和一致性。为了满足这些要求，Fabric 应用了 Gossip 数据分发协议。

8.1.1 Gossip 协议 (Gossip protocol)

节点利用 Gossip 来以一种可扩展的方式广播账本和通道数据。Gossip 出来的消息是连续的，并且通道上的每个节点都在不断地接收当前来自多个节点的账本中已达成一致性的数据。每个通过 Gossip 传输的消息都会被签名，因此由拜占庭节点发送的伪造的消息将会很容易地被识别出来，而且可以防止将消息分发到不希望发送的目标处。节点因为受到延迟、网络分区或者其他原因的影响导致缺少部分区块的情况，最终将通过联系已拥有这些缺失的区块的节点的方式，与当前账本状态进行同步。

基于 Gossip 的数据传播协议在 Fabric 网络上执行三个主要功能：1) 通过不断识别可用的成员节点并最终监测节点离线状态的方式，对节点的发现和通道中的成员进行管理。2) 通过通道中的所有节点来分发账本数据。任何数据未同步的节点都可以通过通道中其他节点来标识缺失的区块，并通过复制正确的数据来进行同步。3) 通过允许点对点状态传输更

新账本数据，使新加入连接的节点快速得到同步。

基于 Gossip 的广播由节点接收来自该通道中的其他节点的消息，然后将这些消息转发到通道上的多个随机选择的节点。这个节点数是个可配置的常数。节点也可以主动拉取消息，而不是等待消息发送。循环重复这个操作，使通道中成员的账本和状态信息不断保持和当前最新状态同步。为了传播新区块，通道中的领导者节点从排序服务中拉取数据，并向其他节点发送 Gossip 消息。

8.1.2 Gossip 消息传输 (Gossip messaging)

在线的节点通过持续地广播“活跃”消息来表明它们的可用性，每条消息都包含公钥基础设施 (PKI) 的 ID 和消息发送者对消息的签名。节点通过收集这些活跃消息来维护通道成员身份。如果没有节点能从某个特定的节点收到活跃消息，那么这个“死亡”的节点最终将从通道成员身份列表中被删除。由于“活跃”信息是通过密码学算法进行签名的，因此恶意节点无法伪装成其他节点，因为它们缺少根证书颁发机构 (CA) 授权的签名密钥。

除了将接收到的消息进行自动转发外，状态协程还会在每个通道上同步节点间的世界状态。每个节点不停地从通道中的其他节点中提取区块，以便在出现差异时修正自己的状态。由于不需要固定连接来维护基于 Gossip 的数据传播，因此该流程可以可靠地为共享账本保证数据的一致性和完整性，包括对节点崩溃的容错。

由于通道之间相互隔离，一个通道上的节点不能在其他任何通道上发送或共享信息。尽管任何节点都可能属于多个通道，但是通过将基于节点通道订阅的机制作为消息分发策略，节点无法将被分隔开的消息传播给不在通道中的节点。



注意 1) 点对点消息的安全性由节点的 TLS 层处理，不需要签名。节点通过其由 CA 分配的证书进行身份验证。节点在 Gossip 层的身份认证会通过 TLS 证书体现。账本中的区块由排序服务进行签名，然后传递给通道中的领导者节点。2) 认证过程由节点的成员管理服务的提供者进行管理。当节点第一次连接到通道中的时候，TLS 会话将与 Fabric 成员身份绑定。这样本质上使每个节点与相连的节点进行认证，从而与网络和通道中的成员身份关联起来。

8.2 Gossip 之服务组件

在这部分文档中，我们将要介绍 Gossip 服务的组件与基本功能。

8.2.1 protos/gossip 分析

在本节中将分析 Gossip 模块中的消息类型的讲解。

1. Gossip 中涉及的消息类型

在 `protos/gossip` 的文件夹下主要有如下文件：

```
message.proto
message.pb.go
extensions.go
```

以及其他一些测试文件。

`message.proto` 中定义了两个 `grpc` 服务，以及 Gossip 服务所使用到的消息类型。`message.pb.go` 是使用 `goprotobuf` 生成的相应代码文件。`extensions.go` 则是 Gossip 原型方法的拓展，用于辅助 Gossip 服务处理消息。我们主要分析一下 `message.proto` 中定义的 `grpc` 服务和包含的消息类型。`message.proto` 中包含的内容主要如下：

```
// 定义了一个 Gossip 服务，其中包含了两个 rpc 服务
service Gossip {

    // GossipStream 是用来发送与接收信息的 grpc stream
    rpc GossipStream (stream Envelope) returns (stream Envelope) {}

    // Ping 是用来探测 remote peer 的存活情况
    rpc Ping (Empty) returns (Empty) {}
}

// Envelope 包括了一个序列化的 GossipMessage 信息
// 以及对该信息的一个签名
// 它也可能包括一个 SecretEnvelope (一个序列化的 Secret)
message Envelope {
    bytes payload = 1;
    bytes signature = 2;
    SecretEnvelope secret_envelope = 3;
}

// SecretEnvelope 是一个序列化的 Secret 以及对该 Secret 的一个签名
message SecretEnvelope {
    bytes payload = 1;
    bytes signature = 2;
}

// Secret 是一个可以从 Envelope 中被忽略的实体
// 当 remote peer 接收了 Envelope 且不应该知道 secret 的内容时，peer 就将 Secret 忽略
message Secret {
    oneof content {
        string internalEndpoint = 1;
    }
}

// GossipMessage 定义了 Gossip 网络中传送的消息
message GossipMessage {
```

```
// 主要用于测试，但将来可能会用于确保消息传递
uint64 nonce = 1;

// 消息的通道
// 一些GossipMessages可能会将其置为nil，因为它们是跨通道的
// 但另一些也可能不会
bytes channel = 2;
```

```
enum Tag {
    UNDEFINED    = 0;
    EMPTY        = 1;
    ORG_ONLY     = 2;
    CHAN_ONLY    = 3;
    CHAN_AND_ORG = 4;
    CHAN_OR_ORG  = 5;
}
```

```
// 决定哪些 peers 可以转发消息
Tag tag = 3;
```

```
oneof content {
    // Membership 成员
    AliveMessage alive_msg = 5;
    MembershipRequest mem_req = 6;
    MembershipResponse mem_res = 7;

    // Contains a ledger block 包含一个账本区块
    DataMessage data_msg = 8;

    // 用来 push&pull
    GossipHello hello = 9;
    DataDigest data_dig = 10;
    DataRequest data_req = 11;
    DataUpdate data_update = 12;

    // 空信息，用来 ping
    Empty empty = 13;

    // ConnEstablish 用来建立连接
    ConnEstablish conn = 14;

    // 用来中继关于 state 的信息
    StateInfo state_info = 15;

    // 用来发送 StateInfo messages 集合
    StateInfoSnapshot state_snapshot = 16;

    //用来请求 StateInfoSnapshots
    StateInfoPullRequest state_info_pull_req = 17;
```



```

// 用来向 remote peer 请求 blocks 集合
RemoteStateRequest state_request = 18;

// 用来向 remote peer 发送 blocks 集合
RemoteStateResponse state_response = 19;

// 用来表明 peer 成为 leader 的意图
LeadershipMessage leadership_msg = 20;

// 用来学习一个 peer 的 certificate
PeerIdentity peer_identity = 21;

Acknowledgement ack = 22;

// 用来请求私有数据
RemotePvtDataRequest privateReq = 23;

// 用来回复私有数据请求
RemotePvtDataResponse privateRes = 24;

// 打包将要分发的私有数据
// 在背书后的私有读写集
PrivateDataMessage private_data = 25;
}
}

```

通过 message.proto 文件我们可以了解到在 Gossip 网络中将会传输的各种消息类型。下一节是对上述各种消息类型的详细定义，在此处不赘述。

2. 消息类型对应的 Tag 与发送方式

前面我们介绍到，在 Gossip 服务中有多种消息类型，每个发送的 gossip message 中都包含了一个 Tag 字段，而 Tag 字段即控制了 message 的发送范围，比如若 message 的 Tag 为 CHAN_AND_ORG，则 message 的发送范围被限定在了同一个 channel 与同一个 Org 中，实际上就是同一个 Org 中。

在 Gossip 中有两种消息发送方式，一种是 gossip，另一种是 end to end。某些消息类型通过 gossip 的方式发送，gossip 方式的特点就在于会根据待发消息的 Tag 字段，选取出有资格接受该消息的 remote peers，然后从 remote peers 中随机选择一些 peers 将消息发送出去，所以 gossip 方式具有随机性。而 end to end，顾名思义，在准备好消息后，也会指定需要接收的 remote peers，然后将消息发送出去，不存在随机性。

下面我们先总结一些各种消息类型对应的 Tag 与发送方式以及每种方式的使用场景，后面我们再对 gossip 发送方式与 end to end 发送方式做详细的介绍。

(1) AliveMessage

❑ Tag: EMPTY。

❑ 发送方式：

○ gossip:

- periodicalSendAlive 方法周期性向外发送 Alive Msg。

(2) MembershipRequest

□ Tag: EMPTY。

□ 发送方式:

○ end to end:

- channel 配置初始化或更新时, 尝试连接配置的 anchor peers 与 bootstrap peers。
- InitiateSync 方法周期性向外界请求成员视图。
- periodicalReconnectToDead 方法周期性尝试连接 dead peer。

(3) MembershipResponse

□ Tag: EMPTY。

□ 发送方式:

- end to end: 在 sendMemResponse 方法中向目标 remote peer 发送成员视图请求回复。

(4) DataMessage

□ Tag: CHAN_AND_ORG。

□ 发送方式:

- gossip: 在 DeliverBlocks 方法中构造区块后分发给同组织中其他 peers。

(5) GossipHello

□ Tag: EMPTY (利用 Pull 机制做视图交换时)。

□ Tag: CHAN_AND_ORG (利用 Pull 机制做组织内区块同步时)。

□ 发送方式:

- end to end: 在 Hello 方法中发给特定的 remote peers。

(6) DataDigest

□ Tag: EMPTY (利用 Pull 机制做视图交换时)。

□ Tag: CHAN_AND_ORG (利用 Pull 机制做组织内区块同步时)。

□ 发送方式:

- end to end: 在 SendDigest 方法中回复给特定的 remote peer。

(7) DataRequest

□ Tag: EMPTY (利用 Pull 机制做视图交换时)。

□ Tag: CHAN_AND_ORG (利用 Pull 机制做组织内区块同步时)。

□ 发送方式:

- end to end: 在 SendReq 方法中发送给特定的 remote peers。

(8) DataUpdate

□ Tag: EMPTY (利用 Pull 机制做视图交换时)。

□ Tag: CHAN_AND_ORG (利用 Pull 机制做组织内区块同步时)。

□ 发送方式:

- end to end: 在 SendRes 方法中回复给特定的 remote peer。

(9) Empty

□ Tag: 不包含任何内容。

□ 发送方式: 通过调用 grpc 标准库中的 Invoke 方法来实现 Ping 功能。

(10) ConnEstablish

□ Tag: EMPTY。

□ 发送方式:

- end to end: 用于 gossip 握手。当一个节点连接另一节点进行握手时, 发送此消息证明自己身份。

(11) StateInfo

□ Tag: CHAN_OR_ORG。

□ 发送方式:

- gossip: 在 publishStateInfo 方法中将最新的 stateInfo Msg 散发出去。

(12) StateInfoSnapshot

□ Tag: CHAN_OR_ORG

□ 发送方式:

- end to end: 在 HandleMessage 方法中调用 ReceivedMessage 的 Respond 方法回复 stateInfo pull 请求。

(13) StateInfoPullRequest

□ Tag: CHAN_OR_ORG。

□ 发送方式:

- end to end: 在 requestStateInfo 方法中向 remote peers 发送 stateInfo pull 请求。

(14) RemoteStateRequest

□ Tag: CHAN_OR_ORG。

□ 发送方式:

- end to end: 在 requestBlocksInRange 方法中向某一 remote peer 发送请求以获取缺失的区块。

(15) RemoteStateResponse

□ Tag: CHAN_OR_ORG。

□ 发送方式:

- end to end: 在 handleStateRequest 方法中调用 ReceivedMessage 的 Respond 方法回复 stateRequest, 向 remote peer 回复其所请求的区块。

(16) LeadershipMessage

□ Tag: CHAN_AND_ORG。

❑ 发送方式:

○ gossip:

- 在 leader 方法中, 若当前 peer 是 Org 的 leader, 则该 peer 不断向外界发送 leader declaration, 声明自己的 leader 身份。
- 在 propose 方法中, 向同一个 Org 内的 remote peers 发送 leadership proposal。

(17) PeerIdentity

❑ Tag: EMPTY。

❑ 发送方式: 。

- gossip: 在 newCertStore 方法中初始化 CertStore 时, 将自身的 identity 信息加入相关缓存中。

(18) Acknowledgement

❑ Tag: 用来回复发送方一个 ack, 无需 Tag 信息。

❑ 发送方式:

- end to end: 在 Ack 方法中调用 ReceivedMessage 的 Respond 方法回复一个 ackMsg。

(19) RemotePvtDataRequest

❑ Tag: CHAN_ONLY。

❑ 发送方式:

- end to end: 在 scatterRequests 方法中向 remote peer 发送私有数据请求。

(20) RemotePvtDataResponse

❑ Tag: CHAN_ONLY。

❑ 发送方式:

- end to end: 在 handleRequest 方法中调用 ReceivedMessage 的 Respond 方法回复私有数据请求。

(21) PrivateDataMessage

❑ Tag: CHAN_ONLY。

❑ 发送方式:

- end to end: 在 SendWithAck 方法中确定发送方式为 sendToEndpoint, 即 end to end 的方式。

8.2.2 Gossip 服务组件

整个 Gossip 服务初始化的入口是在 peer/node/start.go 文件下的 serve 函数中, serve 函数中调用了 gossip/service/gossip_service.go 文件下的 InitGossipService 函数。InitGossipService 函数进一步调用同文件下的 InitGossipServiceCustomDeliveryFactory 函数, 在 InitGossipServiceCustomDeliveryFactory 下执行了 gossipServiceInstance = &gossipServiceImpl{...}, 即是对全局唯一 Gossip 服务实例的初始化。

peer 的全局实例 `gossipServiceInstance`，即代表了 peer 下的 Gossip 服务，`gossipServiceInstance` 是一 `gossipServiceImpl` 类型的结构体，`gossipServiceImpl` 结构体定义在 `gossip/service/gossip_service.go` 文件下，定义如下：

```
type gossipServiceImpl struct {
    gossipSvc
    privateHandlers map[string]privateHandler
    chains           map[string]state.GossipStateProvider
    leaderElection   map[string]election.LeadershipService
    deliveryService  map[string]deliverclient.DeliveryService
    deliveryFactory  DeliveryServiceFactory
    lock             sync.RWMutex
    mcs              api.MessageCryptoService
    peerIdentity     []byte
}
```

我们针对该定义中的各字段对 peer 的 Gossip 服务的各组件进行分析。

1. gossipSvc

`gossipSvc` 是 `gossip_service.go` 文件下定义的变量类型，具体是 `gossip.Gossip`，对应 `gossip/gossip/gossip.go` 文件下的 `type Gossip interface`，而 Gossip 服务的初始化中对 `type Gossip interface` 的实现是 `gossip/gossip/gossip_impl.go` 文件下的 `gossipServiceImpl` 结构体（注意：1）该结构体名称与上面提到的 `gossipServiceInstance` 的实现的结构体名称相同；2）同一接口会有多个结构体对其实现，需要明确此处使用的是哪个结构体实现）。

`gossipSvc` 组件的初始化就在 `InitGossipServiceCustomDeliveryFactory` 函数中 `gossipServiceInstance` 实例初始化代码的上方，通过调用 `gossip/integration/integration.go` 文件下的 `NewGossipComponent` 函数，`NewGossipComponent` 函数首先读取相关配置，然后调用 `gossip/gossip/gossip_impl.go` 文件下的 `NewGossipService` 函数。`NewGossipService` 函数创建了 `gossipServiceImpl` 实例并把它和给定的 gRPC server 绑定在一起。

`gossip.go` 文件下的 `type Gossip interface` 定义如下：

```
type Gossip interface {
    // 将某一 message 发送到 remote peers
    Send(msg *proto.GossipMessage, peers ...*comm.RemotePeer)
    // 将某一 message 发送到所有满足 SendCriteria 的 peers
    SendByCriteria(*proto.SignedGossipMessage, SendCriteria) error
    // 返回被认为还 alive 的 NetworkMembers
    Peers() []discovery.NetworkMember
    // 返回被认为还 alive 并且订阅了给定 channel 的 NetworkMembers
    PeersOfChannel(common.ChainID) []discovery.NetworkMember
    // 更新 peer 发布给其他 peers 的 discovery layer 的 self metadata
    UpdateMetadata(metadata []byte)
    // 更新 peer 发布给其他 peers 的与其 channel-related state 的 self metadata
    UpdateChannelMetadata(metadata []byte, chainID common.ChainID)
```

```

// 发送一个 message 到网络中的其他 peers
Gossip(msg *proto.GossipMessage)
// PeerFilter 接收一个 SubChannelSelectionCriteria 并且返回一个 RoutingFilter
// RoutingFilter 选择那些满足给定 criteria 并且发布了它们的 channel participation 的
// peer identities
PeerFilter(channel common.ChainID, messagePredicate api.SubChannelSelection
Criteria) (filter.RoutingFilter, error)
// Accept 返回一个明确只读的通道, 其中存储的是其他节点发送来的符合某个谓词的 messages
// 如果 passThrough 是 false, 这些 messages 先由 gossip layer 处理
// 如果 passThrough 是 true, gossip layer 不会介入, 并且这些 messages 可以被发送回
// 对应的 sender
Accept(acceptor common.MessageAcceptor, passThrough bool) (<-chan *proto.
GossipMessage, <-chan proto.ReceivedMessage)
// JoinChan 使得一个 Gossip 实例加入 channel
JoinChan(joinMsg api.JoinChannelMessage, chainID common.ChainID)
// LeaveChan 使得一个 Gossip 实例离开一个 channel
// 该 Gossip 实例仍旧可以传播 stateInfo message
// 但是不能再参与进 block 的拉取, 并且不能再返回一个 channel 中所含 peers 的列表
LeaveChan(chainID common.ChainID)
// SuspectPeers 使得一个 Gossip 实例验证被怀疑的 peers 的 identities
// 并且关掉与那些被发现 identities 不合法的 peers 的之间的 connections
SuspectPeers(s api.PeerSuspector)
// 停止该 gossip component
Stop()
}

```

gossip_impl.go 文件下的 gossipServiceImpl 结构体定义如下:

```

type gossipServiceImpl struct {
    selfIdentity      api.PeerIdentityType // 该peer自己的 Identity
    includeIdentityPeriod time.Time             // 自peer启动时将自己的certificate包含在
    // Alive message之中的持续时间, 默认为10s
    certStore         *certStore           // certStore 用来处理与 peer identity
    // 有关的消息
    idMapper           identity.Mapper       // 维护peers的pkiID与certificates之间的
    // 映射
    presumedDead      chan common.PKIIDType // 存储疑似 dead 的 peer 的 pkiID 的通
    // 道, 默认大小为100
    disc              discovery.Discovery   // discovery 模块, 用来维护当前的网络视图
    comm              comm.Comm            // Comm 模块, 用来与同样嵌入了 Comm 模块
    // 的其他 peers 通信
    incTime           time.Time
    selfOrg           api.OrgIdentityType  // peer 自己所属 Org 的 identity
    *comm.ChannelDeMultiplexer // 一个用来接收channel注册 (AddChannel)
    // 与发布 (DeMultiplex) 的结构体
    logger            *logging.Logger
    stopSignal        *sync.WaitGroup
    conf              *Config              // Gossip 服务的相关配置
    toDieChan         chan struct{}        // 用来标识 Gossip 服务是否停止的通道, 默认
    // 大小为1
    stopFlag          int32                // Gossip服务是否停止的标识
}

```



```

emitter          batchingEmitter // emitter模块, 用来发送 message
discAdapter      *discoveryAdapter // discovery适配器, 用来为discovery模块提
                                供在discovery模块中声明的comm接口的所需功能
secAdvisor       api.SecurityAdvisor // SecurityAdvisor定义了一个外部的辅助对象,
                                提供安全和身份相关的功能
chanState        *channelState // 维护 peer 所加入的 channel 的信息
disSecAdap       *discoverySecurityAdapter // discovery安全适配器, 用来验证Alive
                                message等
mcs              api.MessageCryptoService // MessageCryptoService是在gossip组件与peer
                                加密层之间的协议(contract); MessageCrypto
                                Service被gossip组件用来验证与授权远程peer
                                与它们所发送的数据, 也用来验证从Orderering
                                Service收到的区块
stateInfoMsgStore msgstore.MessageStore // 消息存储, 当接收到一个message时, 将message
                                放入一内部缓存
certPuller       pull.Mediator // Mediator是一个包装PullEngine的组件, 它提供
                                了执行pull同步所需的方法
}

```

2. privateHandlers

Gossip 服务的 privateHandlers 字段维护了当前 peer 中每一个 channel 到其 privateHandler 结构体的映射。

3. chains

Gossip 服务的 chains 字段维护了当前 peer 中每一个 channel 到其 GossipStateProvider 的映射。type GossipStateProvider interface 定义在 gossip/state/state.go 文件下, 接口的实现是同文件下的 type GossipStateProviderImpl struct。GossipStateProvider 是通过运行状态复制与发送 missing block 请求到其他节点, 获取能够全部填充一系列缺失的块的接口。

每一个 channel 的 GossipStateProvider 都在 gossip/service/gossip_service.go 下的 InitializeChannel 方法中被初始化, 而 InitializeChannel 方法则进一步调用了 gossip/state/state.go 文件下的 NewGossipStateProvider 函数。真正代表 Gossip 服务每个 channel 的 GossipStateProvider 模块的实例是 gossip/state/state.go 文件下的 GossipStateProviderImpl 结构体类型。

4. leaderElection

如果一个 peer 的 core.yaml 文件中的 peer.gossip.useLeaderElection 配置项为 TRUE, 则该 peer 对于每一个 channel 都会开启一个 leaderElection 模块, 用于领导节点的选举。type LeaderElectionService interface 定义在 gossip/election/election.go 文件下, 该接口的实现是同文件下的 type leaderElectionSvcImpl struct。leaderElection 模块的初始化, 和上一部分中 GossipStateProvider 的初始化一样, 都是在 gossip/service/gossip_service.go 下的 InitializeChannel 方法中进行的。

首先代码会读取 `core.yaml` 文件中的 `peer.gossip.useLeaderElection` 与 `peer.gossip.orgLeader` 配置项。如果 `useLeaderElection` 为 `TRUE`，则执行相应的 `leader` 选举算法选举 `Org` 中的 `leader`；如果 `orgLeader` 为 `TRUE`，则该 `peer` 被静态的定义为该 `Org` 的 `leader peer`；`useLeaderElection` 与 `orgLeader` 若同时为 `TRUE`，则抛出异常；若同时为 `FALSE`，则该 `peer` 不可以连接到 `Ordering Service` 来获取区块。

- ❑ 如果 `useLeaderElection` 为 `TRUE`，则调用当前文件下的 `newLeaderElection Component` 方法，该方法首先获取当前 `peer` 的 `Identity` 并初始化一个 `leader` 选举适配器，然后再进一步调用 `gossip/election/election.go` 文件下的 `NewLeaderElectionService` 函数。`NewLeaderElectionService` 函数根据传入的参数初始化了一个 `leader` 选举服务的实例 `leaderElectionSvcImpl`，`NewLeaderElectionService` 函数执行 `le := &leaderElectionSvcImpl{...}` 将 `leader` 选举服务初始化后，执行 `go le.start()` 开启了一个 `goroutine`，相关选举服务的具体实现通过该 `goroutine` 进行，具体解析见下节。
- ❑ 如果 `orgLeader` 为 `TRUE`，则该 `peer`，即对于当前 `channel` 代表该 `Org` 与 `Ordering Service` 进行通信拉取区块的 `leader peer`，具体是通过调用 `Gossip` 服务中的 `deliveryService` 组件下的 `StartDeliverForChannel` 方法实现的，该方法定位在 `core/deliverservice/deliveryclient.go` 文件下，对该方法的分析见下面对 `Gossip` 服务的 `deliveryService` 组件的分析。

5. deliveryService

`Gossip` 服务的 `deliveryService` 字段维护了当前 `peer` 中每一个 `channel` 到其 `DeliverService` 的映射。`type DeliverService interface` 定义在 `core/deliverservice/deliveryclient.go`，该接口的实现同文件下的 `type deliverServiceImpl struct`。接口定义如下：

```
// DeliverService 用来与 orderers 交流以获得新的区块并将所获区块发送到 committer service
type DeliverService interface {
    // StartDeliverForChannel 动态地将从 Ordering Service 获得的区块发送到 channel peers
    // 当 delivery 结束，finalizer 函数被调用
    StartDeliverForChannel(chainID string, ledgerInfo blocksprovider.LedgerInfo,
        finalizer func()) error

    // StopDeliverForChannel 通过停止 channel 的 block provider 来停止传递区块
    StopDeliverForChannel(chainID string) error

    // UpdateEndpoints 更新所连接的 Ordering Service 的端点
    UpdateEndpoints(chainID string, endpoints []string) error

    // Stop 终止 delivery 服务并关闭 connection
    Stop()
}
```

deliveryService 模块的初始化和上一部分中 leaderElection 的初始化一样，都是在 gossip/service/gossip_service.go 下的 InitializeChannel 方法中进行的。Gossip 服务首先会判断 peer 在该 channel 下是否已经存在了 deliveryService 实例，若不存在，则调用 Gossip 服务中 deliveryFactory 模块下的 Service 方法，新建一个 delivery client 的实例。

在上一节中，若 peer 的 peer.gossip.orgLeader 配置项为 TRUE，则 Gossip 服务调用 deliveryService 组件下的 StartDeliverForChannel 方法。StartDeliverForChannel 是 DeliverService 中最重要的方法，下面我们详细分析一下该方法。

```
func (d *deliverServiceImpl) StartDeliverForChannel(chainID
string, ledgerInfo blocksprovider.LedgerInfo, finalizer func())
error
```

方法首先会判断该 deliverService 实例是否已经停止 (stopping) 或者对于某个 channel 已经存在 blockProvider，若正在停止或者已存在相应的 blockProvider 则返回相应的 error 信息，若既没有正在停止也不存在相应的 blockProvider，则继续向下执行。

- ❑ 首先利用 deliverServiceImpl 的 newClient 方法新生成一个 Client (broadcastClient 结构体类型，该结构体定义在 core/deliverservice/client.go 文件下)，再用 core/deliverservice/blocksprovider/blocksprovider.go 文件下的 NewBlocksProvider 函数，构建新的 blocksprovider 实例。
- ❑ 构建好新的 blocksprovider 实例后，便开启一个 goroutine 调用该实例的 DeliverBlocks 方法，并开始从 Ordering Service 拉取区块。

6. deliveryFactory

deliveryFactory 是用于生成创建并初始化 delivery service 实例的组件，类型为 type DeliveryServiceFactory interface，该接口就定义在 gossip_service.go 文件下，定义如下：

```
type DeliveryServiceFactory interface {
    // 返回一个 delivery client 的实例
    Service(g GossipService, endpoints []string, msc api.MessageCryptoService)
    (deliverclient.DeliverService, error)
}
```

Service 方法的调用即上一小节中我们提到的，是在 deliverService 新建实例的时候调用的。

7. lock

lock 组件即 Gossip 服务实例的读写锁，在 gossip_service.go 文件下的 gossipServiceImpl 结构体的各个方法中，有的使用了写锁定，有的使用了读锁定。

写锁定的使用：

- ❑ InitializeChannel 方法：初始化 gossipServiceImpl 的相关组件。

❑ Stop 方法：停止 Gossip 服务实例。

读锁定的使用：

❑ DistributePrivateData 方法：用于在 channel 中分发私有读写集。

❑ AddPayload 方法：用于将 message payload 添加到给定的链。

8. mcs

mcs 组件的类型是 gossip/api/crypto.go 文件下的 type MessageCryptoService interface，实现是 peer/gossip/mscs.go 文件下的 type mspMessageCryptoService struct。MessageCryptoService 是在 Gossip 服务组件中与 peer 一同被 Gossip 服务组件验证与授权远程 peer 与 peer 们所发送的数据，也用来验证从 Ordering Service 收到的区块。

MessageCryptoService 接口定义如下：

```
type MessageCryptoService interface {

    // GetPKIIdOfCert 返回了 peer 身份的 PKI-ID
    // 如果发生错误，该方法则返回 nil
    // 该方法并不验证 peerIdentity，验证过程应该在 execution flow 中完成
    GetPKIIdOfCert(peerIdentity PeerIdentityType) common.PKIIdType

    // 如果区块已经被正确签名了的话，VerifyBlock 返回 nil
    // 声明的 seqNum 是包含在该 block 的 header 中的序列号
    // 否则返回错误
    VerifyBlock(chainID common.ChainID, seqNum uint64, signedBlock []byte) error

    // Sign 用该 peer 的 signing key 对 msg 签名
    // 如果没错的话则输出该签名
    Sign(msg []byte) ([]byte, error)

    // Verify 使用 peer 的 verification key 来验证该签名
    // 是否是一个合法的签名
    // 如果成功的话则返回 nil
    // 如果 peerIdentity 是空的话则验证失败
    Verify(peerIdentity PeerIdentityType, signature, message []byte) error

    // VerifyByChannel 检查在一个 peer 的 verification key 下
    // 该信息的签名是否有效，在特定通道的上下文中
    // 如果验证成功，Verify 返回 nil
    // 如果 peerIdentity 是空则验证失败
    VerifyByChannel(chainID common.ChainID, peerIdentity PeerIdentityType,
        signature, message []byte) error

    // ValidateIdentity 验证远程 peer 的身份
    // 如果身份无效 / 撤销 / 过期则返回错误
    // 否则返回 nil
}
```

```

ValidateIdentity(peerIdentity PeerIdentityType) error

// Expiration 返回:
// 如果到期了, 返回该 identity 到期的时间, nil
// 如果没有到期, 返回一个零值 time.Time, nil
// 如果不能判断该身份是否会到期, 返回一个零值, error
Expiration(peerIdentity PeerIdentityType) (time.Time, error)
}

```

对 MessageCryptoService 接口进行实现的 mspMessageCryptoService 结构体, 详细定义如下:

```

type mspMessageCryptoService struct {
    channelPolicyManagerGetter policies.ChannelPolicyManagerGetter
    localSigner                crypto.LocalSigner
    deserializer               mgmt.DeserializersManager
}

```

结构体定义中的三个字段, 分别代表不同的功能: channelPolicyManagerGetter, channel 策略管理获得器, 这是一个支持性的接口, 用来获取给定 channel 的策略管理器 (policy manager), localSigner, 本地签名器, 用来执行当前 Gossip 服务的本地签名的, deserializer: 也是一个支持性的接口, 用来获取本地和 channel 的反序列化器

9. secAdv

secAdv 组件类型为 type SecurityAdvisor interface, 该接口定义在 peer/api/channel.go 文件下, 定义如下:

```

// SecurityAdvisor 定义了一个提供安全性和身份相关功能的外部辅助对象
type SecurityAdvisor interface {
    // OrgByPeerIdentity 返回 给定 peer 对应的 OrgIdentityType
    // 出错则返回 nil
    // 此方法不验证 peerIdentity
    // 该验证应该在执行流程期间适当地进行
    OrgByPeerIdentity(PeerIdentityType) OrgIdentityType
}

```

而实现是在 peer/gossip/sa.go 文件下的 type mspSecurityAdvisor struct。对于整个系统的安全而言, 能够动态更新 MSPs 是极为重要的, channel 的 MSPs 是通过 Ordering Service 分发的配置交易 (configuration transaction) 来更新的, secAdv 组件与上一小节中的 mcs 组件, 都是为了实现这个目标提供了相关的功能。

8.2.3 gossip 消息发送方式详解

前面说过, 在 Gossip 服务中有两种消息发送方式, 分别是 gossip 与 end to end, end to end 方式较容易理解, 我们下面详细介绍一下 gossip 方式。gossip_impl.go 文件下的 gossip ServiceImpl 结构体作为 Gossip 服务的核心, 其有一个 emitter 模块, 在 gossipServiceImpl 初始

化时, emitter 模块是通过 `gossip/gossip/batcher.go` 文件下的 `newBatchingEmitter` 方法实例化的。gossip 发送方式的核心就与该模块息息相关, 下面详细介绍一下 emitter 模块使用到的 `gossipBatch` 方法。

`gossipBatch` 方法将要发送的消息分为如下 5 类:

- ❑ `blocks`: 区块消息。
- ❑ `stateInfoMsgs`: 状态信息消息。
- ❑ `orgMsgs`: 组织内消息。
- ❑ `leadershipMsgs`: 领导类消息。
- ❑ `others`: 其他。

`gossipBatch` 从要发送的 `Msgs` 的切片中, 解析出这几类消息, 然后通过设定的过滤条件, 从当前视图中筛选出有资格接收特定种类消息的 `remote peers`, 然后将消息发送。

1. 消息类别解析

`gossipBatch` 中定义了 5 个谓词, 通过定义的谓词, 结合同文件下的 `partitionMessages` 函数实现对消息类别的解析。定义的谓词:

- ❑ `isABlock`: 判断消息类型是否为 `DataMessage`。
- ❑ `isAStateInfoMsg`: 判断消息类型是否为 `StateInfo`。
- ❑ `aliveMsgsWithNoEndpointAndInOurOrg`: 判断该消息是否为 `aliveMsg`, 判断发送该 `aliveMsg` 的 `remote peer` 是否有 `external endpoint`, 判断该 `remote peer` 是否属于当前组织。
- ❑ `isOrgRestricted`: 判断该消息是否满足 `aliveMsgsWithNoEndpointAndInOurOrg` 或者判断该消息是否只应该在组织中传播。
- ❑ `isLeadershipMsg`: 判断该消息类型是否为 `LeadershipMessage`。

`partitionMessages` 接收两个输入参数, 第一个是上面所说的谓词, 第二个是一个消息切片, 该函数最后输出的是两个消息切片, 一个是满足谓词的消息的切片, 另一个是剩余消息的切片。因为上面所说的 5 个谓词, 有些是有包含关系的, 比如说 `isOrgRestricted` 包含了 `isLeadershipMsg`, 故消息类别的解析也是有顺序限制的: `blocks`→`leadershipMsgs`→`stateInfoMsgs`→`orgMsgs`→`others`

2. blocks

对于 `blocks` 消息, 在 `gossipBatch` 方法中调用了同文件下的 `gossipInChan` 方法来发送 `blocks` 消息。`gossipInChan` 方法的两个输入参数分别为: `blocks`——消息的切片和 `remote peers`——过滤器。`remote peers` 过滤器是通过利用 `gossip/filter/filter.go` 文件下的 `CombineRoutingFilters` 函数, 将三个基本过滤器通过逻辑组合而成, 三个基本过滤器分别是:

- ❑ `gc.EligibleForChannel`。判断一个给定的 `peer` 是否有资格获得该 `channel` 下的区块。
- ❑ `gc.IsMemberInChan`。判断一个给定的 `peer` 是否是该 `channel` 的成员。

□ `g.isInMyorg`。判断一个给定的 `peer` 是否属于本 `Org`。

在 `gossipInChan` 方法中, 对于 `blocks` 切片中的各个消息, 提取其所属 `channel` 信息, 将 `blocks` 按所属 `channel` 归类, 对于每一类 `block`, 筛选出有资格接收该 `block` 的 `remote peers`, 并从筛选出的 `remote peers` 中随机选出最多 3 个 (由 `core.yaml` 文件下的 `peer.gossip.PropagatePeerNum` 配置项决定), 将 `block` 发送给这些 `peers`。

特别地, 还会将某一 `msg` 的发送者 (`sender`), 从 `peers` 的待发送列表中过滤掉, 以免造成消息发送循环。

3. leadershipMsgs

对于 `leadershipMsgs` 的处理过程与 `blocks` 消息基本一致, 区别在于, `blocks` 消息最多只会发送给 3 个 `remote peers`, 但对于 `leadershipMsgs` 消息, `gossipInChan` 会将消息发送给所有满足过滤条件的 `remote peers`。这也容易理解, 因为 `leader` 选举与声明是需要 `Org` 中的全体成员参与的。

4. stateInfoMsgs

`stateInfoMsgs` 的处理, 首先要遍历 `stateInfoMsgs`。对于每一条 `stateInfoMsg`, 首先判断给出的 `peer` 是否属于本 `Org`, 然后通过过去发送的 `stateInfoMsgs` 查看 `stateInfo` 所在的 `channel`。如果这个 `channel` 存在且有 `external endpoint`, 则需要判断给出的 `peer` 是否是这个 `channel` 的成员。然后通过 `remote peers` 过滤器筛选出要发送的 `peers`, 将 `stateInfoMsg` 进行发送。三个基本过滤器分别是:

- `g.conf.PropagatePeerNum`。要发送的节点个数 (由 `core.yaml` 文件下的 `peer.gossip.PropagatePeerNum` 配置项决定)。
- `g.disc.GetMembership()`: 网络中存活的成员。
- `peerSelector`: 判断节点是否在 `channel` 上。

5. orgMsgs

通过 `remote peers` 过滤器筛选出要发送的 `peers`, 三个基本过滤器分别是:

- `g.conf.PropagatePeerNum`。要发送的节点个数 (由 `core.yaml` 文件下的 `peer.gossip.PropagatePeerNum` 配置项决定)。
- `g.disc.GetMembership()`。网络中存活的成员。
- `g.isInMyorg`: 判断节点是否在组织内。

然后对 `orgMsgs` 进行遍历, 逐条发送给筛选出的 `peer` 节点。

8.3 Gossip 之服务初始化

Gossip 服务在初始化时, 需要启动不同的 `goroutine` 来实现不同消息的发送及监听。

8.3.1 gossipSvc 组件

在 gossipSvc 初始化的过程中开启了如下 goroutine:

```
// 定期清除过期的 message
- go store.expirationRoutine()
  // 定期将满足条件的 peer 与其 identity(certificate)清除
- go idMapper.periodicalPurgeUnusedIdentities()
  // 监听端口
- go func() {
    defer commInst.stopWG.Done()
    s.Serve(l1)
  }()
  // 周期性地将打包好的 messages 传给回调函数 gossipBatch 以发送
- g.disc = discovery.NewDiscoveryService(...) 中启动的多个 goroutine
  - go p.periodicEmit()
    // 每隔5s向外发送一次 AliveMsg
  - go d.periodicalSendAlive()
    // 每2.5s遍历一次上一次收到的AliveMsg
  - go d.periodicalCheckAlive()
    // 从一个只读通道中读取remote peer发送过来的membership messages并做处理
  - go d.handleMessages()
    // 每25s尝试一次重连已死亡的节点
  - go d.periodicalReconnectToDead()
    // 处理假设已死亡的节点
  - go d.handlePresumedDeadPeers()

  // peers 之间的 identities 信息交流同步
- go func() {
    for !engine.toDie() {
      time.Sleep(sleepTime)
      if engine.toDie() {
        return
      }
      engine.initiatePull()
    }
  }()
- go g.start()
  // 同步 peers 间成员视图
  - go g.syncDiscovery()
    // 处理 dead 的 peer
  - go g.handlePresumedDead()
    // 处理收到的 Messages
  - go g.acceptMessages(incMsgs)
  // 连接配置的 bootstrap remote peers
- go g.connect2BootstrapPeers()
```

下面从第一行命令开始详细介绍命令的具体功能。

```
go store.expirationRoutine()
```

该 goroutine 在 gossipSvc 实例的 stateInfoMsgStore 模块初始化时启动, 启动流程如下: `-g.stateInfoMsgStore = g.newStateInfoMsgStore()` - `return msgstore.NewMessageStoreExpirable(...)` - `go store.expirationRoutine()`

stateInfoMsgStore 模块类型为 gossip/gossip/msgstore/msgs.go 文件下的 type MessageStore interface, 实际实现是同文件下的 type messageStoreImpl struct。该 goroutine 每 4s (与 core.yaml 文件中的 peer.gossip.publishStateInfoInterval 配置项有关) 读取 gossipSvc 实例的 stateInfoMsgStore 模块中的 msg, 然后读取 msg 的创建时间, 与所设置的 msgTTL (同样与 publishStateInfoInterval 配置项有关, 默认为 400s) 做比较, 并把过期的 msg 从 []*msg 中清除出去。

```
go idMapper.periodicalPurgeUnusedIdentities()
```

该 goroutine 在 gossipSvc 实例的 idMapper 模块初始化时启动, 启动流程如下: `-g.idMapper = identity.NewIdentityMapper(...)` - `go idMapper.periodicalPurgeUnusedIdentities()`

idMapper 模块类型为 gossip/identity/identity.go 文件下的 type Mapper interface, 实际实现是同文件下的 type identityMapperImpl struct。该 goroutine 每隔 6 分钟执行一次检查, 对于已经被撤销、过期, 或者长时间未使用 (超过 1 个小时) 的 peer 的 identity (certificate), 将它们从 gossipSvc 的 idMapper 模块 (该模块维护 pkiID 与 peers identities (certificates) 之间的映射) 的 pkiID2Cert 字段键值对中删除。

```
go func() {...}() 监听端口
```

该 goroutine 在 gossipSvc 实例的 comm 模块初始化时启动。

```
go p.periodicEmit()
```

该 goroutine 在 gossipSvc 实例的 emitter 模块初始化时启动, 启动流程: `- g.emitter = newBatchingEmitter(...)` - `go p.periodicEmit()`。

emitter 模块类型为 gossip/gossip/batcher.go 文件下的 type batchingEmitter interface, 实际实现是同文件下的 type batchingEmitterImpl struct。该 goroutine 周期性地执行 emit (发送) 操作 (emit 有两种触发条件, 一种是 gossipServiceImpl 的 emitter 字段中存储的 msg 数量大于配置文件中设定的阈值, 一种是定期的计时器到期), 每隔 10 ms 发送一次, 该配置由 core.yaml 的 peer.gossip.maxPropagationBurst Latency 配置项决定。

另外, 在 gossipSvc 初始化其 emitter 模块时, 给定了 4 个配置, 其中 3 个分别为 peer.gossip.propagateIterations、peer.gossip.maxPropagationBurstSize、peer.gossip.maxPropagationBurstLatency, 分别代表每个 msg batch 被发送的次数、缓存 msg 的最大值、连续两次 push msg 之间的最大时间间隔, 默认值分别为 1 次、10 个、10ms, 每一次将新的 msg 加入 emitter 对应缓存中, 都会判断是否已经达到最大缓存数, 如

果达到了则执行 emit 操作。

emit 操作实际上是将要发送的 msg 打包成 msgs2beEmitted 的切片，然后传送给一个回调函数——即前面所说的第 4 个配置，代码中给定的函数是 gossip/gossip/gossip_impl.go 下的 func (g *gossipServiceImpl) sendGossipBatch(a []interface{})。

func (g *gossipServiceImpl) sendGossipBatch(a []interface{}) 继续调用 func (g *gossipServiceImpl) gossipBatch(msgs []*proto.SignedGossipMessage), gossipBatch 是真正决定将要传送给 (gossip) 到哪些 peers 的方法，对该方法的介绍见前文对 gossipSvc 的解析。

g.disc = discovery.NewDiscoveryService(...) 中会启动多个 goroutine。

gossipSvc 实例在初始化其 disc 模块时开启了多个 goroutine，启动流程：
- g.disc = discovery.NewDiscoveryService - gossip/discovery/discovery_impl.go 文件下 NewDiscoveryService 函数来启动多个 goroutine：

```
- go d.periodicalSendAlive()
- go d.periodicalCheckAlive()
- go d.handleMessages()
- go d.periodicalReconnectToDead()
- go d.handlePresumedDeadPeers()
```

disc 模块类型为 gossip/discovery/discovery.go 文件下的 type batching Emitter interface，实际实现是 gossip/discovery/discovery_impl.go 文件下的 type gossipDiscoveryImpl struct。

```
go d.periodicalSendAlive()
```

该 goroutine 定期地（由 core.yaml 文件中的 peer.gossip.aliveTimeInterval 配置项决定，默认为 5s）向外发送一次 AliveMsg，发送 AliveMsg 是通过调用 gossipSvc 实例的 comm 模块下的 Gossip(msg *proto.SignedGossipMessage) 方法实现的，而该方法实际最终是通过调用 gossip/gossip/batcher.go 文件下的 func (p *batchingEmitterImpl) Add(message interface{}) 方法，下一步的发送由 emitter 完成。

```
go d.periodicalCheckAlive()
```

该 goroutine 定期地（与 core.yaml 文件中的 peer.gossip.aliveExpirationTimeout 配置项有关，默认为 2.5s）遍历一次上一次收到的 AliveMsg。如果某个 remote peer 发过来的最新的 AliveMsg 距现在已经过去了 25 秒（同样由 aliveExpirationTimeout 配置项决定），则将该 remote peer 从 disc 模块下的 aliveMembership 字段实例（类型为：map[string]*timestamp）中移除，放入 deadMembership 字段实例（类型同 aliveMembership 字段）中。

```
go d.handleMessages()
```

该 goroutine 调用 disc 模块下的 comm 模块中的 Accept() <-chan proto.Received Message 方法, 获得一个只读通道, 不断循环读取通道中的内容, 并对读取到的 message 做处理。disc 模块只处理 3 种类型的消息, 分别是 Alive Message、MembershipResponse Message 和 MembershipRequest Message。handleMessages 从只读通道中读取消息数据后, 调用同文件下的 handleMsgFromComm 方法对消息进行处理。

Alive Message 是一个 peer 发送给 remote peer 以告知其自身的存在与活动的消息类型:

```
type AliveMessage struct {
    Membership *Member
    Timestamp  *PeerTime
    Identity   []byte
}
```

MembershipRequest Message 是用来从一个 remote peer 请求其成员信息的消息类型:

```
type MembershipRequest struct {
    SelfInformation *Envelope
    Known           []byte
}
```

MembershipResponse Message 是用来回复 MembershipRequest 的消息类型:

```
type MembershipResponse struct {
    Alive []*Envelope
    Dead  []*Envelope
}
```

```
go d.periodicalReconnectToDead()
```

该 goroutine 检查自己维护的 dead member 的视图。对于每一个被怀疑已 dead 的 remote peer, 都开一个 goroutine, 利用 disc 模块下的 comm 模块中的 Ping(peer *NetworkMember) bool 方法探测一下该 remote peer 是否可达, 如果可达则向其发送 MembershipRequest 消息, 请求成员信息。等到上述 goroutine 全都执行完毕之后, 等待 25s (与 core.yaml 文件中的 peer.gossip.reconnectInterval 配置项有关, 默认为 25s) 再开始下一次尝试连接。

```
go d.handlePresumedDeadPeers()
```

该 goroutine 不断调用 disc 模块下 comm 模块中的 PresumedDead() <-chan common.PKIDType 方法, disc 模块下的 comm 模块即 gossipSvc 实例的 discAdapter 模块, PresumedDead 方法返回的是 gossipSvc 实例的 presumedDead 模块 (gossipSvc 实例的 presumedDead 模块是一个通道, 默认大小为 100, 用来存储已 dead 的 remote peer), 遍历其中存储的 remote peer, 将其添加进 disc 模块的 deadMembership 模块维护的键值对中, 从 disc 模块的 aliveMembership 模块维护的键值对中删除。

```
go func() { ... }() peers 之间的 identitis 信息交流同步
```

该 goroutine 在 gossipSvc 实例初始化其 certPuller 模块时启动, 启动流程:

```
❑ g.certPuller = g.createCertStorePuller()
❑ pull.NewPullMediator(conf, adapter)
    ○ p.engine = algo.NewPullEngineWithFilter(...)
    ○ go func(){...}
```

certPuller 模块为 gossip/gossip/pull/pullstore.go 文件下的 type Mediator interface 类型, 实际实现是同文件下的 type pullMediatorImpl struct。Mediator 是一个包装了 PullEngine 的组件, 并提供了执行 pull 同步所需的一些方法, Mediator 对于特定的消息类型配置了相应的钩子 (hook) 来对该消息做处理, 在 pull 同步中共有四种消息类型: hello、digest、req、res。

PullEngine 是执行基于 pull 的同步操作的对象, 并维护由字符串号码标识的 items 的内部状态。基于 pull 的同步协议如下所示:

- 1) Initiator 发送一个 Hello Message, 附上一个特定的 NONCE 到一系列的 remote peers;
- 2) 每个 remote peers 返回一个带有它们所持 items 摘要 (digest) 的 message 并附上收到的 Hello Message 中的 NONCE;
- 3) Initiator 验证接收到的 NONCES, 聚合收到的 digest, 构造包含 item id 的请求, 并将 req 消息 (依旧带上了 NONCE) 发送往特定的 remote peer 以请求该 item;
- 4) remote peers 收到 req 消息后, 就将包含 req 消息中所请求的 item 的 res 消息 (依旧带上了 NONCE) 发送回 Initiator。

Other peer	Initiator
O <----- Hello <NONCE> -----	O
/ \ <----- Digest <[3,5,8, 10...], NONCE> ----->	/ \
<----- Request <[3,8], NONCE> -----	
/ \ <----- Response <[item3, item8], NONCE>----->	/ \

从模块的名称可以看出该模块的功能, 即该模块可以同步 peers 之间各个 peer 所持有 (自己的或其他 peer) 的 identity (certificate)。gossipSvc 的 certPuller 模块与 certStore 模块是搭配使用的, 在 certStore 模块初始化时, 执行 g.certStore = newCertStore(g.certPuller, g.idMapper, selfIdentity, mcs), 可以看到刚刚初始化了的 certPuller 模块是 certStore 模块的一部分。在 gossip/gossip/certstore.go 文件下的 newCertStore 函数下, 执行了这样一句代码: puller.Add(selfIDMsg), 即该 peer 将自己的 pkiID 与 identity 信息存储进了 gossipSvc 实例下的 certStore 模块下的 certPuller 模块。所以我们可以知道, 在 certPuller 模块下执行的基于 pull 的同步, 同步的对象是各个 peer 自己所知的 peers 的 identity (certificate) 信息。

该 goroutine 会执行 gossip/gossip/algo/pull.go 文件下的 func (engine *PullEngine) initiatePull() 方法, 每执行一次该方法代表一轮 pull 同步操作, 每当执行完一次后再过一段时间执行下一次 (与 core.yaml 文件中的 peer.gossip.PullInterval 配置项有关, 默认为 4s)。

initiatePull 方法首先会向已知的所有 peer 发送 Hello Message, 等待一个 digestWaitTime

后 (与 core.yaml 文件中的 peer.gossip.digestWaitTime 配置项有关, 默认为 1s), 执行同文件下的 processIncomingDigests 方法。

processIncomingDigests 方法会处理接收到的 request Message, 然后回复 response Message。等待一个 responseWaitTime 后 (与 core.yaml 文件中的 peer.gossip.responseWaitTime 配置项有关, 默认为 2s), 执行同文件下的 endPull 方法, 清空 certPuller 模块实例中存储的 item2owners、peers2nonces、nonces2peers 等信息, 完成这一轮的 pull 同步。

```
go g.start()
```

该 goroutine 可以视为 gossipSvc 实例的开启, 在该 goroutine 下开启了几个重要的 goroutine, 如下所示:

```
- go g.syncDiscovery()
- go g.handlePresumedDead()
- go g.acceptMessages(incMsgs)
```

下面我们对这几个 goroutine 分别进行分析。

```
go g.syncDiscovery()
```

该 goroutine 会执行同文件下的 syncDiscovery 方法, 该方法会调用 gossip/discovery/discovery.go 文件下的 InitiateSync(peerNum int) 方法, 实际实现是在 gossip/discovery/discovery_impl.go 文件下。InitiateSync 方法会请求一些 peers 的成员视图 (membership view), 每当执行完一次后, 间隔 4s (与 core.yaml 文件下的 peer.gossip.PullInterval 配置项有关, 默认为 4s) 再执行下一次。该 goroutine 实现了 peers 之间成员视图的交换。

```
go g.handlePresumedDead()
```

该 goroutine 会不断地读取 gossipSvc 实例中 comm 模块中 PresumedDead() <-chan common.PKIidType 方法返回的只读通道, 从该通道中读取出的数据是已被认为 dead 的 peer 的 pkiID。读取后放入 gossipSvc 实例的 presumedDead 模块中, 该模块也是一个通道, gossipSvc 实例初始化时默认设置该通道的大小为 100。

```
go g.acceptMessages(incMsgs)
```

在该 goroutine 开启之前, start 方法首先定义了一个 msgSelector, 调用了 gossipSvc 实例中 comm 模块下的 Accept(common.MessageAcceptor) <-chan proto.ReceivedMessage 方法, 该方法返回一个只读通道 incMsgs。goroutine go g.acceptMessages(incMsgs) 不断从只读通道 incMsgs 中读取到来的 Message, 再调用同文件下的 handleMessage 方法进行处理。

gossip/gossip/gossip_impl.go 文件下的 handleMessage 方法对接收到的 ReceivedMessage 做处理, 将接收到的消息分为三类: 1) 受 channel 限制的 Message; 2)

Discovery 类消息，包括：Alive Message、MembershipRequest Message、MembershipResponse Message；3）基于 pull 机制的消息，并且该消息是 identity 相关消息。

```
go g.connect2BootstrapPeers()
```

该 goroutine 会读取 core.yaml 文件下的 peer.gossip.bootstrap 配置项，里面包含了配置 remote peers 的 endpoint，每一个 peer 启动时都会尝试连接 bootstrap 配置项中配置的 remote peers。连接的过程包括：构建一个 identifier（身份识别器），再通过 gossipSvc 的 disc 模块下的 Connect(member NetworkMember, id identifier) 方法建立连接。

构建 identifier 包括如下操作：首先进行 Handshake 获取 remote peers 的 identity，再检测是否与当前 peer 同属一个 Org（期望同属一个 Org，否则返回错误信息），再通过 remote peer 的 identity 在 gossipSvc 实例的 mcs 模块中查询其 pkiID（期望能够查询到，否则返回错误信息），最后返回对 remote peer 的身份认证信息（PeerIdentification）与 error 信息。

8.3.2 chains 组件

Gossip 服务中 chains 组件的每一个 channel 对应的 GossipStateProvider 初始化时，都会开启四个 goroutine，如下：

```
// 监听传入的通信
- go s.listen()
  // 对传入的 block 相关的 message 进行处理
- go s.deliverPayloads()
  // 执行反熵以填补缺失的区块
- go s.antiEntropy()
  // 处理状态请求消息
- go s.processStateRequests()
```

下面对上述代码进行详细地解释说明。

```
go s.listen()
```

该 goroutine 不断尝试读取 chains 组件对应当前 channel 下的 GossipStateProviderImpl 结构体实例中的 gossipChan 字段与 commChan 字段，两个字段均为只读通道类型。

1. 若从 gossipChan 通道中读取到新消息，意味着从 gossip channel 中收到了新的 message，则进一步开启一个 goroutine go s.queueNewMessage(msg)，将收到的 message 交给同文件下的 queueNewMessage 方法进行处理。一般收到的 message 类型是 DataMessage，这种消息类型包含有一个 block。queueNewMessage 方法在进行必要的有效性检测后，调用同文件下的 addPayload 方法，以非阻塞模式将 message 中包含的 block 加入一个区块缓存（block buffer）中。

2. 若从 commChan 通道中读取到新消息，意味着从其他 remote peers 收到了新的 message，则进一步开启一个 goroutine go s.dispatch(msg)，将收到的 message 交给同文件下的

dispatch 方法进行处理。一般收到的 message 类型是：

(1) 状态同步 (state synchronization) 相关信息。包括 RemoteStateRequest、RemoteStateResponse。RemoteStateRequest 用来向 remote peers 请求一系列 blocks 的消息类型，RemoteStateResponse 是用来向 remote peers 发送一系列 blocks 的消息类型。

(2) 私有数据 (private data) 相关消息。包括 RemotePvtDataRequest、RemotePvtDataResponse、PrivateDataMessage。RemotePvtDataRequest 用来请求错失的私有读写集 (private rwset)，RemotePvtDataResponse 用来回复私有数据复制请求 (private data replication request)，PrivateDataMessage 是交易背书后包含私有数据信息的消息类型。

```
go s.deliverPayloads()
```

该 goroutine 不断执行一个 for 循环，其中包含有 select 语句，select 语句中的一个 case 是 `case <-s.payloads.Ready()`，该 case 意味着下一个区块已经到来。在上文中我们说到，新的 DataMessage 到来之后，会交给 addPayload 方法处理，将 message 中包含的 block 加入区块缓存中，但同时 addPayload 方法还会往 GossipStateProviderImpl 实例下的 payloads 字段代表的结构体实例下的 readyChan 字段（类型是一个缓存为 0 的通道）中写入一个空结构体，所以上面的 `case <-s.payloads.Ready()` 读取到的就是该通道。获知新的 message 到达后，该 goroutine 调用对数据进行一些相关处理，然后调用同文件下的 commitBlock 方法，提交新的 block。

提交 block 的具体过程：之前说到 Gossip 服务的 chains 组件下的每一条 channel 对应的 GossipStateProviderImpl 实例通过 commitBlock 方法提交区块。commitBlock 方法做了两部分工作：1) 调用 StoreBlock 方法（实际实现在 gossip/privdata/coordinator.go 文件下），提交了 block 与可用的私有交易信息；2) 调用 UpdateChannelMetadata 方法（实际实现在 gossip/gossip/gossip_impl.go 文件下），更新了与该 channel 相关的状态信息（状态信息周期性发送到其他 remote peers）

block 的存储过程还是比较复杂的，具体体现在 StoreBlock 方法中，包括获取私有数据读写集的过程，只有顺利完成后才能调用 core/ledger/ 下的代码存储区块。

```
go s.antiEntropy()
```

反熵，顾名思义是降低混乱程度。该 goroutine 周期性地（每隔 10s，与全局变量 defAntiEntropyInterval 有关）执行一次如下操作：1) 获取自身账本的高度 current_high；2) 迭代地获取该通道所有可用的 (alive) remote peers，获取它们的账本高度信息 max_high；3) 如果 `current_high < max_high`，则调用同文件下的 requestBlocksInRange 方法请求缺失的区块。

peers 之间同步账本区块主要靠 gossip/state/state.go 文件下的 func (s *GossipStateProviderImpl) requestBlocksInRange(start uint64, end uint64) 方法完成，我们主要分析一下这个方法。

如果 $\text{end} - \text{start} \geq 10$ (与全局变量 `defAntiEntropyBatchSize` 有关), 则每次构造的 `GossipMessage` 中请求的 block 数目为 10 个, 剩余的不足 10 个在最后一个 `GossipMessage` 中请求。

调用同文件下的 `stateRequestMessage` 方法, 构造状态请求信息, 重点在于构造的 `GossipMessage` 中的 `Content` 字段如下:

```
Content: &proto.GossipMessage_StateRequest{
  StateRequest: &proto.RemoteStateRequest{
    StartSeqNum: beginSeq,
    EndSeqNum:   endSeq,
  }
}
```

上面的代码指明了请求的 block 的范围。

发送 `GossipMessage` 请求区块时, 会从当前 channel 中的所有可用的 (alive) remote peers 中, 随机在所有拥有区块高度满足要求 (即拥有区块高度大于等于请求区块高度) 的 peers 中选择一个, 调用 `GossipStateProviderImpl` 实例下的 `mediator` 字段中的 `Send` 方法发送请求。发送完 `GossipMessage` 后进入一段 `select` 代码段, 等待计时器到期或者请求回复到达。在计时器到期之前对一个区块范围的请求最多尝试 3 次 (由全局变量 `defAntiEntropyMaxRetries` 决定), 计时器计时为 3s (由全局变量 `defAntiEntropyStateResponseTimeout` 决定)。对请求回复消息的辨认是通过之前提到过的 `NONCE` 机制实现的。

```
go s.processStateRequests()
```

该 goroutine 通过 `for` 循环与 `select` 语句, 不断监听 `GossipStateProviderImpl` 实例下的 `stateRequestCh` 通道 (该通道默认大小为 100), 若有新的状态请求 (state request) 信息进入, 则调用同文件下的 `handleStateRequest` 方法处理请求信息。

收到新的状态请求信息 `msg` 后, 首先判断请求中请求区块的范围是否有效 (包括一次最多请求 10 个区块等限制), 自身所持有的区块高度是否能满足 `msg` 中请求的区块高度等, 一系列验证通过后便可开始构造回复信息 (类型为 `&proto.GossipMessage`)。重点在于回复信息是通过 `msg` 中的 `Respond` 方法 (实际实现在 `gossip/comm/msg.go` 文件下) 回复的, `Respond` 方法可以将 `GossipMessage` 发送回该消息发出的源头。

8.3.3 leaderElection 组件

`leaderElection` 组件用于领导节点的选举, 将接收到的信息在节点间扩散。

```
go le.start()
```

之前提到, 如果 `core.yaml` 文件中的 `peer.gossip.useLeaderElection` 配置项为 `TRUE`, 则当前 peer 开启一个 leader 选举的服务实例, 实例类型为 `gossip/election/election.go` 文件下的 `leaderElectionSvcImpl` 结构体类型。

该服务实例初始化后开启 goroutine `go le.start()`, 该 goroutine 执行的就是 leader

选举算法。

然后又执行了如下操作：

```
go le.handleMessages() //监听 leader 选举相关消息
le.waitForMembershipStabilization(getStartupGracePeriod()) //等待成员视图稳定
go le.run() //执行 leader 选举并广播相关的 proposal 或 declaration 消息
```

Leader 选举算法性质：1) peers 之间靠 pkiID 分辨各自的身份；2) 每个 peer 不是 leader 就是 follower，算法的目的在于对所有具有相同成员视图的 peers，只选出一个 leader；3) 如果网络分区成 2 个或多个子集，则 leader 的数目等于分区的数目，但是当网络分区又归为一个时，有一个 leader 可以留下来

算法伪代码：

变量：

```
leaderKnown = False
```

不变量：

```
peer监听来自remote peers的消息，当peer收到一个leadership声明(declaration)时，将
leaderKnown设为True
```

Startup()：

```
等待成员视图稳定，或者接收到一个 leadership 声明，或者计时器到期
goto SteadyState()
```

SteadyState()：

```
while true:
```

```
    If leaderKnown is false:
```

```
        LeaderElection()
```

```
    If you are the leader:
```

```
        Broadcast leadership declaration
```

```
        If a leadership declaration was received from
```

```
        a peer with a lower ID,
```

```
        become a follower
```

```
    Else, you're a follower:
```

```
        If haven't received a leadership declaration within
```

```
        a time threshold:
```

```
            set leaderKnown to false
```

LeaderElection()：

```
Gossip leadership proposal message
```

```
Collect messages from other peers sent within a time period
```

```
If received a leadership declaration:
```

```
    return
```

```
Iterate over all proposal messages collected.
```

```
If a proposal message from a peer with an ID lower
```

```
than yourself was received, return.
```

```
Else, declare yourself a leader
```

关于等待成员视图稳定，代码中是按如下方法实现的：首先设定一个计时器，时间长

度为 15s (由 core.yaml 文件中的 peer.gossip.election.startupGracePeriod 配置项决定)。在计时器没到期之前, 每隔 1s (由 core.yaml 文件中的 peer.gossip.election.membershipSampleInterval 配置项决定), 做如下检查: 1) 成员视图是否稳定, 即所掌握的 remote peers 的数目和上一次检查时的数目是否相同; 2) 计时器是否到期; 3) 是否已经有 leader 存在。若上述三个条件任意一个满足“是”, 则结束对成员视图稳定的等待, 开始执行 `go le.run()`。

另外, 如果一个 peer 被选为 leader, 则它每 5s (与 core.yaml 文件中的 peer.gossip.election.leaderAliveThreshold 配置项有关, 设定为配置值的一半) 向外发送一次 leadership declaration。

如果一个 peer 发现, 当前已存在 leader, 且自己不是 leader, 则首先将 leader 是否存在的标志位 `leaderExists` 设为 0, 然后等待 10s (由 core.yaml 文件中的 peer.gossip.election.leaderAliveThreshold 配置项决定), 在这期间如果接收到了其他 leader 发来的 leadership declaration 消息, 则将 `leaderExists` 设为 1。这样做的目的在于, 探测网络分区或 leader 节点是否失效。

```
go le.handleMessages()
```

此函数主要是为了开启处理消息服务。首先, 由适配器 `adapter` 获取接收消息的频道——`msgChan := le.adapter.Accept()`。`msgChan` 是专门处理 `election` 模块中特定消息类型 `msgImpl` 的, 所以 `Accept()` 中新启了一个 goroutine 去把 gossip 服务中获取的 `proto.GossipMessage` 类型的数据转成 `msgImpl` 类型的消息并转发到了 `msgChan` 中。然后, 在 `for` 循环中持续接收 `msgChan` 中到来的一条条消息, 交由 `handleMessage` 函数处理, 处理的逻辑是: 1) 如果消息是 `proposal`, 则记录到 `election` 模块成员 `proposals` 中, 即把所有其他节点送来的想要成为 leader 的“自荐信”先放起来。2) 如果消息是 `declaration`, 则把已经存在 leader 的标识 `leaderExists` 置为 1, 然后若此时模块正在休眠且 `interruptChan` 是空的, 则向 `interruptChan` 发空命令唤醒模块, 若自己当前是 leader 而 `declaration` 中的新 leader 又不是自己, 则调用 `le.stopBeingLeader()` 停止当 leader。这里的 `interruptChan` 可以理解为模块的中断休眠频道, 模块成员 `sleeping` 标识着模块当前是否在休眠, 而 `waitForInterrupt(time)` 函数即通过 `select{...}` 让模块进入休眠, 等待 `time` 时间后或 `declaration` 消息一旦到来, 再置 `sleeping` 为 `false`, 即唤醒模块。

```
waitorMembershipStabilization
```

此函数等待成员关系固定化。这里的成员关系指的是网络中存在的所有活着的节点数量, 固化指成员关系是稳定的。这个等待的过程是: 确定当前时间为 `N` 后开始判断, 使用适配器获取当前成员数量 `viewSize` (追溯适配器可知, 获取成员数量使用的还是 `discovery` 模块), 然后进入 `for` 循环不断地判断成员关系是否稳定, 判断的标准是, 每隔一秒重新获取最新的成员数量 `newSize`, 若 `newSize == viewSize`, 则称当前成员关系是稳定的,

否则，更新 `viewSize` 为最新发现的成员数量，即 `viewSize = newSize`，进行下一轮判断。若整个判断过程超过 $N+15s$ 的时间点（ $15s$ 是由 `core.yaml` 中的 `peer.gossip.election.startupGracePeriod` 指定），或 `leader` 已经出现（即接收到 `declaration` 消息），则停止判断。这里需要深入理解，在一个新初始化的 `chain` 中，新的节点陆续加入到网络中，被 `gossip` 服务的 `discovery` 模块发现并记录，这个过程还是很快的，可能几毫秒的时间内就会发现若干个新节点，所以在判断成员关系是否稳定时，若在等待 $1s$ 后新获取的成员数量还是等于原来的成员数量，那么 `election` 模块就认为在 $1s$ “这么长”的一段时间内都没有新节点加入，则“自认为”当前网络中所有活的节点都已经被发现了，即成员关系固定了。另外，这个等待成员关系固定化的过程是一个辅助 `election` 模块进行选举 `leader` 的，不能无限期待下去而耽误了选举的正事儿，所以规定了 $15s$ ，要是 $15s$ 还没固定，那就不等了。自然的，要是在这期间已经有了 `leader`，那也就没有选举的必要了，没选举的必要，更没再等待固定化的必要了，所以这种情况下会停止判断。

```
go le.run()
```

在前面的基础上，现在可以进行 `election` 模块的任务。1) 如果现在还没有 `leader`，则调用 `le.leaderElection()` 进行选举，这里与其说是选举，不如说是进行自荐，在 `leaderElection()` 中，调用 `le.propose()` 向其他节点传播自己的“自荐信”（即包含自己身份信息的 `proposal` 消息，其他节点收到后会在它们的 `handleMessage` 中存储这封信）。然后调用 `waitForInterrupt(time)` 让模块进入休眠，这里的 `time` 为 $5s$ ，由 `core.yaml` 中 `peer.gossip.election.leaderElectionDuration` 指定，在这个休眠的过程中既接收其他节点发来的它们的“自荐信”并存储在 `proposals` 中，也等待可能出现的 `declaration` 消息。在模块被唤醒后，判断此时是否有 `leader` 存在，若存在（说明在休眠期间接收到了 `declaration` 消息），则自己放弃成为 `leader`；若不存在，则拿自己的身份与 `proposals` 中已经收到“自荐信”中的其他节点的身份进行一一对比，看看自己是否比它们都更有资格当 `leader`。这里所说的身份是一个节点的 `PKI-ID`，而判断谁更有资格当 `leader` 的标准是 `bytes.Compare(peerID(id), le.id)`，即谁的身份的二进制值更小，谁更有资格。不过 `gossip` 里面的 `leader` 工作量更大，并且没有什么其他的特权和优待。若 `proposals` 中所有的身份都没有自己的身份小，那么自己当选 `leader`，通过调用 `le.beLeader()`，真正成为了一名 `leader`：把已经有 `leader` 和自己是 `leader` 的标识，即成员 `leaderExists` 和 `isLeader`，都置为 1 ，然后调用 `le.callback(true)` 这个“倒钩”函数做 `leader` 在 `gossip` 中该做的事情。2) 选举后，如果自己是 `leader`，则执行 `le.leader()`，如果自己是 `follower`，则执行 `le.follower()`，都是自己成为的角色该做的事情。`leader` 的任务：循环地每休眠 $5s$ 后（因为自身是 `leader`，所有不可能再从其他节点中接收到 `declaration` 消息而使休眠中断）调用 `le.adapter.Gossip(...)` 向其他节点传播发送 `declaration` 消息来宣告自己是 `leader`；`follower` 的任务：清空“信箱” `proposals` 以备下一轮选举，置 `leaderExists` 为 0 （用以等接收到 `leader` 发来的 `declaration` 消息后再置为 1 ），然后用 `select{...}` 等待 $10s$ ，这个

10s 由 core.yaml 中的 `peer.gossip.election.leaderAliveThreshold` 指定, 10s 结束后如果还没收到 leader 发来的 `declaration` 消息, 即 `leaderExists` 还为 0, 则该节点将再发起新一轮的选举。这里仍然是, follower 认为 10s 这么长的时间足够 leader 发送一条 `declaration` 消息给自己, 若 10s 过后还没有收到, 则 follower “自认为” 这个 leader 已死, 则自己将发起新一轮的选举。3) 第 1) 和 2) 是在 `run()` 中循环进行的。

当一个节点调用 `le.beLeader()` 真正成为了一名 leader 时, 调用了 `le.callback(true)`, 当一个节点调用 `le.stopBeingLeader()` 停止作 leader 时, 调用了 `le.callback(false)`, 两者只是参数不同。“倒钩” 函数 `callback` 由 `election` 模块初始化时由参数传入, 具体的赋给的值是 `service/gossip_service.go` 中的 `onStatusChangeFactory(...)` 返回的函数 `func(isLeader bool) {...}`, 即根据是否是 leader 而分别执行 `g.deliveryService.StartDeliverForChannel` 和 `g.deliveryService.StopDeliverForChannel`, 从这点就可以看出, leader 多干了 `DeliverFor Channel` 向其他节点分发数据相关的一些事情, 这与 `ordering` 服务的数据分发客户端有关。

State

`state` 模块原型为 `GossipStateProviderImpl`, 在 `state/state.go` 中定义, 相关代码都在 `state` 目录中, 可以将其理解为 `gossip` 传播和提交状态消息的管理模块, 这里的状态的意思与账本数据、`block` 数据的意思一致。`state` 模块也封装了一个适配器, `election` 模块类似, 也是被赋值为 `gossip` 服务器实例, 在 `state` 模块中, 消息处理流程如图 8-1 所示:

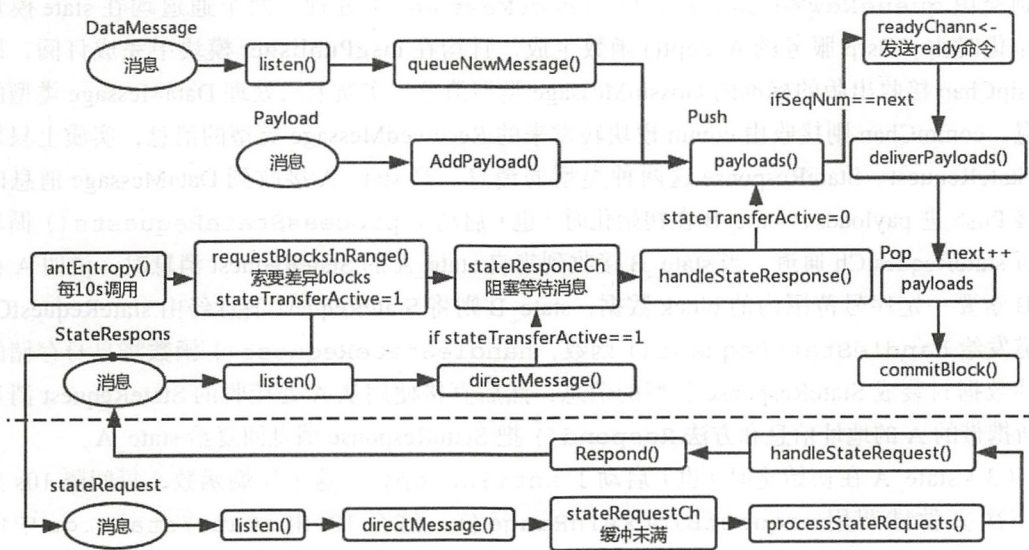


图 8-1 消息处理流程

参看流程图, 虚线以上, 箭头为流的是节点 A 的 `state` 模块 `state_A`, 蓝色以下, 箭头为流的是节点 B 的 `state` 模块 `state_B`。`state` 模块处理三种消息类型: `DataMessage`、

StateResponse 和 StateRequest。在 state 模块专用初始化函数 NewGossipStateProvider 中，初始化 GossipStateProviderImpl 对象后新启动了 4 个 goroutine 来开始运行 state 模块的服务。这些均在流程图中有所反映：

(1) payloads 既是一个存储器，也是一个消息排序输出器，原型为 PayloadsBufferImpl，在 state/payloads_buffer.go 中定义。state_A 把接收的所有消息 Push 进 payloads，一旦所接收到消息的序号 SeqNum == next (next 指期望的下一个消息序号，消息序号即是一个频道的状态高度，也即一个频道的 block 高度)，则向 readyChan 通道发送 ready 命令以指示所期望的消息已经收到，即当前阶段的数据已经准备好了。这里有一个前提 Q 是：ordering 服务输出的消息都是排序过并依次输出给 state 模块的。比如，在 Push() 函数中，若 next 为 2，则说明序号为 1 的消息之前肯定已经被接收并处理过，当前想要的是接收序列号为 2 的消息，在前提 Q 下，state 模块再收到消息的序号也只可能是 1 或 2，因为 ordering 服务可能因某些原因会重发已经发过的消息 1，但绝不可能跳过序号为 2 的消息而去发序号为 3 的消息。因此，若 Push 进的消息的序号小于 2，则直接返回，若等于 2，则存储进成员 buf 后发送 ready 命令。此外，Pop() 函数从 payloads 中弹出一个消息给 state 模块处理，同时将 next 增 1。比如一旦序号为 2 的消息被弹出交由 state 模块处理，则 next 变为 3，即此时序号为 2 的消息已在处理且 payloads 所期望的消息变为了序号为 3 的消息。如此，由前提 Q、Push()、Pop() 三者相互配合，实现 payloads 的消息排序输出功能。

(2) 启动的 listen() 循环接收来自 gossipChan、commChan 通道的消息，接收后分别交由 queueNewMessage() 和 directMessage() 处理。两个通道均在 state 模块初始化时由 gossip 服务的 Accept() 函数生成，且均在 msgPublisher 模块中完成订阅，即 gossipChan 接收出版的标准的 GossipMessage 类型消息，实质上只处理 DataMessage 类型的消息，commChan 则接收由 comm 模块转发来的 ReceivedMessage 类型的消息，实质上只处理 StateRequest、StateResponse 这两种类型的消息。当 state_A 接收到 DataMessage 消息时直接 Push 进 payloads；state_B 在初始化时（也）启动了 processStateRequests() 循环监听 stateRequestCh 通道，当 state_B 接收到来自 state_A 的 StateRequest 消息时，说明 A 在向 B 索要一定序号范围内的 block 数据，state_B 则将 StateRequest 消息经由 stateRequestCh 通道发给 handleStateRequest() 函数，handleStateRequest() 函数将自身存储的这些数据封装成 StateResponse 类型的消息，然后直接使用从 A 处接收的 StateRequest 消息中所携带的 A 的地址信息和方法 Respond() 把 StateResponse 消息回复给 state_A。

(3) state_A 在初始化时（也）启动了 antiEntropy() 这个反熵函数，每间隔 10s 执行一次去尝试调用 requestBlocksInRange()，这个 10s 由 state/state.go 中的 defAntiEntropyInterval 常量指定。反熵的本意就是降低一个事物的紊乱程度，在这里做的就是尝试降低自己节点与其他节点之间状态的差异，即存储的数据高度的差异。由于消息在网络中传播快慢有所不同，自然，每个节点间当前所存储处理的数据也不同，可能 state_B 已经在处理完了序列号为 10 的消息，而 state_A 才刚刚处理完序列号为 7 的消

息。这时 10s 的周期到了, `state_A` 调用 `committer.LedgerHeight()` 获取当前自身账本中的 `block` 高度 `current` (假设为 7), 然后调用 `maxAvailableLedgerHeight()` 获取频道中其他节点中最最高的高度 `max` (假设为 10), 发现 `current` 与 `max` 差了 8~10 的三个数据块, 则调用 `requestBlocksInRange()` 来弥补这个差异。差异弥补了, 也就降低了整个频道状态的紊乱程度, 即反熵。在 `requestBlocksInRange()` 函数中, 会尝试 3 次分批索要。3 次由 `defAntiEntropyMaxRetries` 常量指定, 分批则是在差异较大时, 如几十块数据时, 以 10 块 (由 `defAntiEntropyBatchSize` 常量指定) 为一批进行一次次地索要, 当然这里我们只差 3 块, 索要一次即可。索要时, 先将 `stateTransferActive` 置为 1 以表明自己发生了索要行为, 然后调用 `stateRequestMessage()` 生成 `StateRequest` 消息 `gossipMsg`, 再调用 `selectPeerToRequestFrom()` 随机从满足条件 (即拥有 8~10 这三块数据) 的节点中选出一个索要对象 `peer` (假设 `peer` 为节点 B), 接着向调用 `gossip.Send(gossipMsg, peer)` 向 `state_B` 发送 `StateRequest` 消息后, 进入 `stateResponseCh` 通道阻塞, 以等待 `state_B` 的回复。当 `state_A` 的 `listen()` 监听到 `state_B` 回复的 `StateResponse` 消息时, 把消息交给 `directMessage()` 分流, 此时若 `stateTransferActive==1`, 说明自己索要过 `StateResponse` 消息且当前正在等待这个回复 (否则就说明 `state_A` 之前不存在索要行为也没有在等, 也即当前 `state_A` 与其他节点状态暂时不存在差异, 即便别人主动给 `state_A` 发了, `state_A` 也不会处理), 则再将 `StateResponse` 消息分流至 `stateResponseCh` 通道, 此时阻塞等待结束, `StateResponse` 消息被交由给 `handleStateResponse()` 处理, 处理的过程无非就是把收到的消息中包含的自己索要的没有的块数据 `Push` 到 `payloads` 中以供进一步的处理。`antiEntropy()` 解决的是防止一个节点永远也跟不上其他节点的节奏或越来越落后于其他节点的状态。比如 `state_A` 所在环境导致处理消息的效率本身就很慢, `state_B` 所在环境导致处理消息的效率较快, 两者的环境一直处于不变的状态, 则若没有这个反熵函数, 则 `state_A` 与 `state_B` 的差距会越来越大。

(4) `state_A` 在初始化时 (也) 启动了 `deliverPayloads()`, `deliverPayloads()` 函数循环等待来自 `readyChan` 通道的 `ready` 命令。在以上 (2) (3) 两点 `state_A` 接收到的包含数据的消息, 最后都 `Push` 进 `payloads` 中, 当 `payloads` 处于准备就绪的状态时, 会发送 `ready` 命令, `deliverPayloads()` 函数一旦收到这个命令, 就会将 `payloads` 中存储的所有消息 (一般只有一条) `Pop` 出来, `Unmarshal()` 后将数据封装成 `common.Block` 类型的 `rawBlock`, 最后调用 `commitBlock(rawBlock)`, 一方面将数据提交至自己的核心账册——`committer.Commit(block)`, 另一方面更新自己的状态 (高度)——`UpdateChannelMetadata(...)`。这里的 `committer` 是核心代码中的数据提交模块, 作用就是将数据提交至账本然后记录下来。

8.3.4 gossip 服务的停止

`gossip` 服务, 包括它控制的各个模块的停止, 都是通过一个停止通道来实现。以

gossip 服务对象 `gossipServiceImpl` 为例，它的停止通道是成员 `toDieChan`。在停止服务函数 `Stop()` 中，按顺序依次停止各个模块中，`g.toDieChan <- struct{}{}` 语句就是向停止通道发送一个空命令，而正在服务的 `periodicalIdentityValidation`，`handlePresumedDead`，`acceptMessages`，`Accept` 函数中所起的 `goroutine`，`select` 接收到 `toDieChan` 频道发来的停止服务的空命令，随机 `return` 退出循环而结束 `goroutine`，即停止了服务。gossip 服务中的其他模块也遵循这种停止自身服务的模式。

8.4 Gossip 之消息广播

Fabric 主要利用 Gossip 协议进行消息的广播，保证消息最终的一致性。

8.4.1 gossip 服务消息的散播过程

需要说明的是，首先，这里的消息指的是 `ordering` 服务经 `deliverservice` 服务发送给 gossip 服务器后在众节点间散播的 block 块消息（即把 block 作为 payload 封装到 `DataMessage` 类型的 `GossipMessage`），而 gossip 服务器中处理的其他类型的消息，如 `state` 消息、关系消息等，都具有辅助性质，即都是为了 block 块消息能够在众节点间散播并达到最终一致所服务的。其次，预设的情景是：Fabric 网络中存在一个 ID 为 `chainID` 的频道，该频道包含一个名为 `orgID` 的组织和一个名为 `ordererID` 的 `ordering` 服务节点，`orgID` 组织中包含 ID 为 `nodeA`、`nodeB`、`nodeC`、`nodeD` 的四个节点，每个节点的 gossip 服务器都已初始化，`nodeA` 被指定或暂时选举为 `leader`。留意这些预设对象的名字，下文中将直接使用。再者，消息传播基于的 `grpc` 服务在下文所述的 `deliverservice` 模块初始化中应该还尚未与服务器端建立连接，这里只是假设已连接。最后，在讲述消息散播的过程中，重点是消息如何散播。

相关目录：

```
/fabric/core/deliverservice
/fabric/gossip
/fabric/protos/gossip
```

8.4.2 消息从何而来

一块块 block 消息由 `ordering` 服务序列化后，使用 `deliverservice` 服务的分发客户端发送给 gossip 服务器，即消息直接来自 `deliverservice` 模块。这里假设 `deliverservice` 模块会从 `ordering` 服务依次接收序号在 11~20 之间共 10 块的 block。在传播过程中，这些数据在各个模块间传送，会变化或被封装成不同的消息类型，但均用 `M11`、`M12`...`M20` 表示。

1. deliverservice 模块的初始化

`deliverservice` 模块代码集中在 `deliverservice` 目录下，原型为 `deliverServiceImpl`，在 `deliveryclient.go` 中定义，利用成员 `blockProviders` 对每一个频道 `BlocksProvider` 对象进行

管理。BlocksProvider 对象利用 grpc 客户端从 ordererID 节点接收消息后使用 gossip 服务器开始传送消息，原型为 blocksProviderImpl，在 blocksprovider/blocksprovider.go 中定义。

因为只有 leader 节点才会启动该模块，因此以 nodeA 节点为例。在 start.go 的 serve 中，InitGossipService() 实例化了 gossip 服务器，但此时并未初始化 deliverservice 模块，直到后文的 Initialize() 才间接在 /fabric/core/peer/peer.go 的 createChain() 中调用 service.GetGossipService().InitializeChannel() 将 deliverservice 模块初始化：在 InitializeChannel() 中，调用 g.deliveryFactory.Service(...) 将 gossip 服务器实例和指定的 ordererID 的 IP 地址等封入配置 Config 后传入 deliverservice 模块中，然后调用 StartDeliverForChannel() 启动模块。

2. deliverservice 模块的启动

在 StartDeliverForChannel() 中，启动步骤如下：

(1) 在 blockProviders 中若属于 chainID 的 BlocksProvider 存在，则表明当前 nodeA 的 deliverservice 模块已经在运行 (StopDeliverForChannel())，一旦 nodeA 停止模块，则把 BlocksProvider 从 blockProviders 中删除，则直接返回。

(2) 若不存在，调用 newClient 新建一个 grpc 客户端 client。这个客户端原型为 broadcastClient，在 client.go 中定义，实现的是 /fabirc/protos/orderer/ab.pb.go 中定义的 AtomicBroadcast_DeliverClient 这个 Deliver 服务的 grpc 流客户端接口。这里不直接使用 ab.pb.go 中的 atomicBroadcastDeliverClient，自然是因为这个自动生成的结构不能满足功能的需要。而且可以臆测一下，Deliver 服务的 grpc 流服务端应该实现在 ordering 服务中。

(3) 调用 NewBlocksProvider(...) 新建一个属于 chainID 的 BlocksProvider 对象，并把 2 中新建的 client、deliverservice 模块 Config 中的 gossip 服务器，以及签名对象 CryptoSvc 传给这个对象的各个成员，然后把这个对象放入 blockProviders 中。

(4) 新创建一个 goroutine，运行 go d.blockProviders[chainID].DeliverBlocks()，执行 (3) 中新建的 BlocksProvider 对象的 DeliverBlocks() 服务。

(5) DeliverBlocks() 就是实际办事的函数，在这个函数中，循环使用 client 客户端从 ordering 节点接收 11 块 block 消息，然后使用 switch-case 根据消息的类型分别处理每块消息。这里只关注 DeliverResponse_Block 类型的消息，即我们要传播的原始的 block 块数据。在此分支中，先调用 createPayload() 将 block 数据包装成可存储在本地账本的 payload，再调用 createGossipMsg() 将 payload 包装成可用于传播的 DataMessage 类型的 GossipMessage 消息 gossipMsg (gossipMsg 的 Tag 值为 CHAN_AND_ORG)，然后调用 gossip 服务器的 AddPayload() 直接将 payload 存储在本地的账本中并更新 chanState 模块中对应 chainID 的 channel 对象的状态 (此状态指 channel 实例成员 stateInfoMsg，包含当前从 payload 抽取的高度和时间戳)，再调用 gossip 服务器的 Gossip() 将 gossipMsg 散播出去。而 gossipMsg 被传播到其他节点，比如 nodeB 后，目的也是把 gossipMsg 中包含的 payload 抽取出来后存储到 nodeB 本地的账本中。

8.4.3 消息如何散播

本节讲解在 Fabric 网络中 Gossip 消息是如何散播的。

(1) 序列号为 11~20 的 gossipMsg 进入 nodeA 的 gossip 服务器的 Gossip(gossipMsg) 中, 先把 gossipMsg 包装成 SignedGossipMessage 类型的 sMsg, 然后使用 MessageCryptoService 对 sMsg 签名, 使 sMsg 包含了 nodeA 的身份信息。

(2) 检查 gossipMsg 的 Tag, 运行判断语句 if msg.IsChannelRestricted(), 若 gossipMsg 指定可以在频道范围内散播, 则 nodeA 在 chanState 模块中使用 chainID 对应的 channel 对象的 AddToMsgStore(sMsg) 函数, 将 sMsg 存储在 blockMsgStore 中。在 blocksPuller 中的 itemID2Msg 中以 PKI-ID 为 key 也存储了 sMsg, 并把 sMsg 的序号在 engine 中的 state 中存储一份。这里提一句, 在 blocksPuller 中存储 sMsg 和 sMsg 的序号是为了 pull 消息时使用, 但是在 blockMsgStore 存储一份是为了什么目前笔者还没搞清楚, 因为事实上只见往 blockMsgStore 中存了但是压根没找有在哪里使用。

(3) g.emitter.Add(sMsg), 将 sMsg 包装成 batchedMessage 后添加到 emitter 模块的 buff 中, 准备发送。我们知道 emitter 是一批批发送的, 指定一批大小默认为 10 的 burstSize, 当序号为 20 的 sMsg 被 Add 进 emitter 后, 就会触发 emitter 模块的 emit() 函数。emit() 将现存的消息, 即序列号为 11~20 的消息抽取出 data 并放入“发射数组”msgs2beEmitted, 然后先调用倒钩发送函数 cb(msgs2beEmitted), 再调用 decrementCounters() 清除 emitter 模块中剩余发送次数为 0 的消息, 因为将 sMsg 包装成 batchedMessage 时所给的发送次数默认为 1, 因此这发送过的 10 条 block 消息都会从 buff 中删除。这里需要说明的是, 在同一时段, emitter 模块不一定只收到 block 消息, 也会收到其他类型的、其他频道 ID 的消息 (比如 channel 模块会通过它的适配器往 emitter 中 Add 消息), 而因为我们的关注点在 block 消息, 所以我们这里假设 emitter 只是清一色地接收到了序号为 11~20 这 10 个属于 chainID 的 block 消息。

(4) 倒钩发送函数 cb 即为 gossip 服务的 sendGossipBatch(...), 简单地将接收的 10 条消息数组重新置换成 SignedGossipMessage 格式的消息数组 msgs2Gossip, 然后直接调用 gossipBatch(msgs2Gossip) 发送消息数组。

(5) gossipBatch(msgs[]) 可处理多种类型数据, 这里我们还是只关注 block 消息。调用 partitionMessages(isABlock, msgs) 函数, 将消息数组中的 DataMessage 类型 SignedGossip Message 消息过滤出来放入 blocks, 然后将 blocks 和一个过滤器 filter 传入 gossipInChan(...), 这里的这个过滤器 filter 将在下文单独介绍。

(6) 在 gossipInChan(...) 中, 先调用 extractChannels(messages) 将消息中所有的频道 ID 抽取出来放入 totalChannels, 然后用第一层 for 循环遍历 totalChannels, 针对每个不同频道将属于各自频道的消息在频道内传播, 这里 10 条消息中包含的频道消息都是 chainID, 因此第一层 for 循环唯一一次循环就是针对 chainID 的。接着在第一层循环 for 中, 再次用 partitionMessages 来选出属于 chainID 的消息并放入 messagesOfChannel (这里是 10 条消息

都是属于 chainID 的), 这个是最最终要发送的消息清单。然后获取 chanState 模块管理的对应 chainID 的 channel 模块 gc 供下两步使用。再利用 discovery 模块获取当前 chainID 频道中所有活着的节点 membership (这里就是 nodeB、nodeC、nodeD 三个节点)。再调用过滤器 filter 模块的 SelectPeers(...) 从 membership 中随机筛选出 3 个节点集合 peers2Send, 这里原代码的逻辑会把 nodeB、nodeC、nodeD 三个节点都筛选出来, 但为了体现 gossip 散播的过程, 虽然三个节点都满足条件但只随机选出一个节点, 假设为 nodeB, 这个是最最终要发送的节点清单。清单的原型是 RemotePeer, 包含一个节点的 Endpoint 和 PKIID。最后使用第二层 for 循环遍历消息清单中的所有消息 (即序号为 11~20 的 block 消息), 依次调用 comm 模块的 g.comm.Send(msg, peers2Send...), 向节点清单发送 M11、M12...M20。

(7) comm 模块就是 nodeA 向其他节点传播消息的 grpc 通信的模块, 既是 grpc 流客户端, 也是 grpc 流服务端。站在 nodeA 的角度看, 在发送消息时, 使用的是流客户端。在 Send() 中, for 循环遍历节点清单中的每个节点, 这里只有 nodeB, 针对 nodeB 启动一个 goroutine 来调用 c.sendToEndpoint(peer, msg)。

(8) 在 c.sendToEndpoint(peer, msg) 中, 当参数 peer 值为 nodeB 时, nodeA 先调用 connStore 模块的 connStore.getConnection(peer) 获取当前 nodeB 的 grpc 连接 conn, 当这个连接不存在时会进行创建, 创建的时候会同时运行这个连接的读写函数。conn 即 nodeA 与 nodeB 间的连接, 然后调用 conn 的 conn.send(msg, disconnectOnError) 将消息封装成 msgSending 类型后经 conn 的发送通道 outBuff 从 nodeA 发给 nodeB。经过第 6 步中的第二层循环, 将 M11、M12...M20 最终都会发送给 nodeB。以下步骤以 M11 为例。

(9) nodeB 的 comm 模块作为 grpc 服务器端, 在 GossipStream() 函数中接收到 nodeA 发来的 M11。也即从这一步起, 就要站在 nodeB 的角度来看代码了。nodeB 此刻作为服务器角色, 调用 connStore.onConnected() 获取服务端流与 nodeA 的连接 conn 后, 启动了 conn 的 serviceConnection(), 即新启了 goroutine 来用 readFromStream 接收 nodeA 发来的 M11, 然后抽取 M11 中的 SignedGossipMessage 类型内容作为 msg 后经由 msgChan 发给 conn.handler(msg) 这个 conn 中的“倒钩”成员来处理。

(10) conn 的 handler() 是在上一步的 GossipStream() 中获取 conn 后被赋值的, 该“倒钩”成员所做的是: 把接收的 SignedGossipMessage 类型的 M11 封装成 ReceivedMessageImpl 消息, 交由 msgPublisher 模块的 DeMultiplex() 进行出版。在 nodeB 初始化自己的 gossip 服务对象实例时, 已经在 gossip/gossip_impl.go 的 start() 中调用 incMsgs := g.comm.Accept(msgSelector) 在 msgPublisher 模块中注册订阅了专用于接收 ReceivedMessageImpl 类型消息的频道 incMsgs, 然后又调用了 go g.acceptMessages(incMsgs) 新启了一个 goroutine 来接收 incMsgs 这个通道的消息。DeMultiplex() 出版的 M11 会通过 incMsgs 这个通道发送到了 acceptMessages(incMsgs) 中, 其一旦收到 M11, 会交由 g.handleMessage(msg) 处理。

(11) 在 g.handleMessage(m) 中, 先重新抽取 ReceivedMessageImpl 类型的 M11 中的 SignedGossipMessage 作为消息 msg 进行一系列 if 判断, 调用 g.chanState.lookupChannel

ForMsg(m) 获取 chanState 模块中 chainID 对应的 channel 对象 gc, 经过一系列判断, 会调用 gc.HandleMessage(m) 对接收的原消息 M11 进行处理。

(12) ReceivedMessageImpl 格式的 M11 再次进入 channel 模块, 在 HandleMessage() 中, 先从 ReceivedMessageImpl 类型的 M11 中抽取出 SignedGossipMessage 作为消息 m。在 m 是 DataMessage 的前提下, m 会进入 if m.IsDataMsg() 分支, 进行如下处理: ① gc.blockMsgStore.Add(), 将 M11 存储到 blockMsgStore 中, 如果添加成功, 则继续, 否则直接退出。② gc.Gossip(), 从 nodeB 出发, 从第 1 步开始, 继续在网络中散播 M11。③ gc.DeMultiplex(m), 出版 M11, 供本地订阅接收。④ blocksPuller.Add(), 添加到 blocksPuller, 供 blocksPuller 在 pull 机制中运作 (blocksPuller 中有了 M11, 就不必再向其他节点索要了)。

(13) state 模块在初始化时, 就向 msgPublisher 模块注册订阅了专用于接收 DataMessage 为 Content 的 SignedGossipMessage 消息的通道 gossipChan, 并启动一个 goroutine 来执行 listen(), 接收 gossipChan 中来的消息。因此当发行 M11 时, state 模块会通过 listen() 从 gossipChan 通道中接收 M11 并进行 go s.queueNewMessage(msg) 处理。

(14) 在 queueNewMessage(msg) 中, 先在 M11 中抽取出 payload, 然后 Push 进 payload, 接着会被 payloads 弹出来后交由 commitBlock, 同样, 也是提交到 nodeB 自身的账本中后, 更新 nodeB 自身的 channel 对象的 stateInfoMsg, 再次触发 nodeB 的 channel 的 publishStateInfo, 向其他节点 (包括 nodeA) 发送 StateInfo 消息 (用于向这些节点报备自己的身份, 可结合下文理解报备的意思)。

散播过程中如何选择散播节点

我们做过比喻, gossip 算法类似于办公室八卦信息和疫情传染, 因此一个节点向另一些节点散播消息, 这里的“另一些”在选择上有两个特征: 数量不定; (2) 随机并就近选择。

gossip 在选择节点时使用的是 filter/filter.go, 即 filter 模块。主力函数是 SelectPeers, 辅助函数是 CombineRoutingFilters 和 util 下的 GetRandomIndices。而上述的两个特征, 会由 SelectPeers 和 GetRandomIndices 体现。

上一小节中第 5 步中所提及的过滤器 filter, 指下面这个调用 g.gossipInChan 的第二个参数:

```
//在gossip/gossip_impl.go中的gossipBatch函数中
g.gossipInChan(
    blocks,
    func(gc channel.GossipChannel) filter.RoutingFilter {
        return filter.CombineRoutingFilters(
            gc.EligibleForChannel,
            gc.IsMemberInChan,
            g.isInMyorg)
    })
```

过滤器 filter 在 gossipInChan 中散播 block 时, 作为调用的 filter.SelectPeers 的第 3 个

参数, 被用于筛选适合的节点集合 `peers2Send`。上面代码中展示的筛选条件很清晰: 以一个 `NetworkMember` 形式的节点身份传入 `gc.EligibleForChannel`、`gc.IsMemberInChan`、`g.isInMyorg` 验证并都返回为 `true`, 这个节点才会被选中。撇开后两个不谈, 单说 `gc.EligibleForChannel`, 该条件验证了一个节点的 PKIID 是否存在于 `nodeA` 的 `channel` 对象 (指 `chanState` 模块所管理的对应 `chainID` 的 `channel` 对象, 下同) 成员 `stateInfoMsgStore` 中。这个成员存储节点间发送的 `StateInfo` 消息和消息中携带了节点身份信息。也就是说, `nodeA` 此刻传播 `block` 时, 只有 `stateInfoMsgStore` 中存在的节点才会被筛选出来。而一个节点的身份信息想出现在 `nodeA` 的 `stateInfoMsgStore` 中, 必须在 `nodeA` 开始筛选节点之前, 就把自己的 `StateInfo` 消息通过 `publishStateInfo` 发送给 `nodeA` (`publishStateInfo` 是 `channel` 模块周期性执行的函数), 而 `publishStateInfo` 也是像上述过程那样一步步散播 `StateInfo` 消息的。说到底, 散播使用的是 `comm` 模块实现 `grpc` 网络传输服务, 而只有与 `nodeA` 近的节点, 才能更快地通过网络将自己的 `StateInfo` 消息散播给 `nodeA`, 如此的话, 当 `nodeA` 在散播 `block` 时, 能筛选出的都是离自己特别近或者传输效率特别高的节点。另外, `stateInfoMsgStore` 存储的身份会定时清理, 实际上是每隔 400s 就清理一次 `MessageStore`, 同时用 `callback` 同步清理 `MembershipStore`, 这点可以从 `NewMessageStoreExpirable` 的实现看出。这么做是因为, 一个节点, 如 `nodeA`, 若不实施定时清除其他节点发来的 `StateInfo` 消息, 经过一段时间后, `nodeA` 也会接收到距离较远的节点发来的 `StateInfo` 消息并记录在 `stateInfoMsgStore` 中, 这样当 `nodeA` 筛选节点发送 `block` 时, 这些较远的节点也有可能被选中, 这就违法了就近的原则。

每个节点, 如 `nodeB`, 其 `state` 模块在初始化时都会调用 `UpdateChannelMetadata` 来更新 (虚假更新, 因为给的 `StateInfo` 消息中的高度是现有高度 -1) 一下自己 `channel` 对象成员 `stateInfoMsg`, 并置 `shouldGossipStateInfo` 为 1, 目的就在于驱动 `publishStateInfo` 传播一次这条虚假的 `stateInfoMsg`, 以让距自己比较近的节点, 如 `nodeA` 的 `stateInfoMsgStore` 存储到自己的身份信息, 进而在 `nodeA` 传播 `block` 时自己能在传播的范围之内。

关于虚假更新, 这里解释为: 进行一个虚假更新, 只是为了让距自己比较近的节点存储到自己的身份信息, 在之后传播消息的时候能算自己一份。其实只要这个虚假更新所更新的高度比现有高度低就行, 即 -2、-3 其实也行, 但是因为一个节点最低的高度就是 1 (即 `genesis` 块), 也因为高度没有负数一说, 所有这里给的是 -1。不过这个好像也不太对, 因为既然是虚假更新, 那为什么不直接给出当前的高度呢?

以上就是对 `gossip` 算法就近特征的叙述。

随机特征比较好理解, 由 `GetRandomIndices` 实现, 在 `filter.SelectPeers` 中被调用。

数量不定特征在 `gossip` 中不明显, 因为指定数量的是 `filter.SelectPeers` 的第 1 个参数, 而这个参数在实现中又是由配置指定的。但是当符合条件的节点少于指定的个数时, 则数量不定, 比如配置指定的是每次向 10 个节点散播, 但筛选出来的只有 5 个, 那只能向这 5 个节点散播, 如果筛选出来 8 个, 则只向这 8 个节点散播。

state 模块订阅了 DataMessage 类型数据, 而且是 gossipMessage 或 signedgossipmessage 类型的。

8.4.4 消息去往何方

通过上一节所述可知, 消息在散播进一个节点后, 一方面会存储到自己的节点的账本和一些本地模块中, 另一方面会继续在网络中散播。至于散播何时停止, 见图 8-2 所示。

A、B、C、D 四个节点属于同一频道同一组织。这里模拟一下传播的过程, 初始条件:

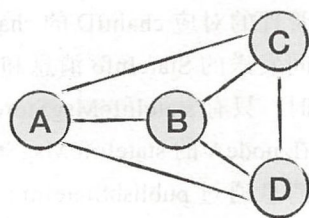


图 8-2 消息传递图

- A-B、B-C、B-D、C-D 均为一个单位距离, A-C、A-D 均为两个单位距离。
- 四个节点处理消息效率一致, 每 100s 才能向一个单位距离范围的节点传播一条消息。
- 每隔 400s, 每个节点的 stateInfoMsgStore 清理一次名单。对于存在时间 ≥ 400 s 的名单, 将废止。
- 只传播一条消息 M, M 是高度为 2 的 block。时间用 T_n 表示, 如 T_{100} 表示时间在 100s 时, T_{110} 表示时间在 110s 时。每个节点发送自身的状态信息用 $S(i, h, t)$ 表示, i 代表身份, h 代表当前账本的高度, t 代表时间戳。每个节点的账本高度当前均为 1。每个节点需花费 10s 才能将 M 存储到自身的账本中。
- 此刻从 0s 开始计时, A 为 leader, 开始接收 deliverservice 发来的消息并开始传播。此前已花费了 100s 时间供四个节点同时初始化, 每个节点此刻只存储在一个单位距离范围的节点名单中, 即 A 中有 $S(B, 1, 0)$; B 中有 $S(A, 1, 0)$, $S(C, 1, 0)$, $S(D, 1, 0)$; C 中有 $S(B, 1, 0)$, $S(D, 1, 0)$; D 中有 $S(B, 1, 0)$, $S(C, 1, 0)$ 。
- 为 T_{10} 时, A 将 M 提交到自己的账本中, 发送 $S(A, 2, 10)$ 给 B, 随即将 M 传播给 B。
- 为 T_{110} 时, B 收到 $S(A, 2, 10)$ 并更新自己的名单, 同时收到 M, 立即传播给 A、C、D。为 T_{120} 时, B 将 M 提交到自己的账本中, 向 A、C、D 发送 $S(B, 2, 120)$ 。
- 为 T_{200} 时, 在 T_0 前 100s 就开始传播 S 的 A-C, A-D 两条 2 个单位距离线路的双方也都收到了各自的 S, 即 A 收到 $S(C, 1, -100)$ 、 $S(D, 1, -100)$; C 收到 $S(A, 1, -100)$; D 收到 $S(A, 1, -100)$, 此刻每个节点的名单中都包含另外三个节点。为 T_{210} 时, A、C、D 收到 B 发送的 M, A 已有 M, 未作进一步处理, C 和 D 立即向其余三个节点传播 M。为 T_{220} 时, C 和 D 将 M 提交到自己的账本中, 向其余三个节点分别发送了 $S(C, 2, 220)$ 、 $S(D, 2, 220)$ 这两个消息, 同时, A、C、D 也收到 B 发送来的 $S(B, 2, 120)$ 消息。为 T_{220} 时, 名单情况为: A 中有 $S(B, 2, 120)$, $S(C, 1, -100)$, $S(D, 1, -100)$; B 中有 $S(A, 2, 10)$, $S(C, 1, 0)$, $S(D, 1, 0)$; C 中有 $S(A, 1, -100)$, $S(B, 2, 120)$, $S(D, 1, 0)$; D 中有 $S(A, 1, -100)$, $S(B, 2, 120)$, $S(C, 1, 0)$ 。

□ 为 T310 时, A、B、D 收到 C 发来的 M, 因为都有了, 未作进一步处理; A、B、C 都收到 D 发来的 M, 因为都有了, 未作进一步处理。为 T320 时, A、B、D 收到 C 发来的 S (C, 2, 220); A、B、C 收到 D 发来的 S (D, 2, 220), 各自更新自己的名单。为 T320 时, 名单情况为: A 中有 S (B, 2, 120), S (C, 2, 220), S (D, 2, 220); B 中有 S (A, 2, 10), S (C, 2, 220), S (D, 2, 220); C 中有 S (A, 1, -100), S (B, 2, 120), S (D, 2, 220); D 中有 S (A, 1, -100), S (B, 2, 120), S (C, 2, 220)。

□ 为 T400 时, 每个节点的 stateInfoMsgStore 清理一次名单, C、D 中的 S (A, 1, -100) 将被清除。至此 M 在四个节点间散播过程终止。

从这里可以看出用文字描述散播过程还是相当费力的, 本想画一个类似 state 模块那样的通信流程图的, 但发现很难画清晰。上述过程还是最简单的情况, 但是依然可以看出, 对 stateInfoMsgStore 清理名单的周期进行一定量的设置, 每个节点所持有的散播名单都会保持在一定范围内。

8.5 channel 通道的设计与实现

8.5.1 概述

channel 在 fabric 中是一个相当重要的概念。channel 本身存在于 orderer 节点内部, 但需要通过 peer 节点使用 peer channel ... 命令进行维护。一个 peer 节点要想与另一个 peer 节点发生交易, 最基本的前提就是两个节点必须同时处在同一个 channel 中, block 账本与 Channel 也是一对一的关系, 即一个 channel 一个账本。Channel 的基本动作如表 8-1 所示。

表 8-1 channel 的基本动作

动 作	释 义
create	在 orderer 节点内部创建一个 channel, 如: peer channel create -o orderer.example.com:7050 -c mychannel -f ./channel.tx
join	加入一个 channel, 如: peer channel join -b mychannel.block
update	升级 channel 的某一组织的配置, 如: peer channel update -o orderer.example.com:7050 -c mychannel -f ./Org1MSPanchors.tx
list	列出当前系统中已经存在的所有由 peer 节点创建的 channel
fetch	获取 channel 中的 newest、oldest 块数据或当前最新的配置数据, 如: peer channel fetch config config_block.pb -o orderer.example.com:7050 -c mychannel

chaincode 分为 SCC 和 ACC, 这里 channel 也分为 system channel 和 application channel。system channel 是随着 orderer 节点运行之时根据 genesis.block 创建的, 而通过 peer channel ... 维护的 channel, 均为 application channel。对 application channel 发起维护命令的 peer 节

点必须是提交的配置文件（channel.tx, mychannel.block, Org1MSPanchors.tx）中所配置的组织中的一员，如 peer0 执行 create，则 peer0 必须是 channel.tx 中所配置的组织中的一员，而 channel.tx 对应生成的 block 消息，就相当于 application channel 中的 genesis.block。这里所说的属于组织中的一员，其实是指该 peer 节点要持有该组织所颁发的证书。

8.5.2 配置文件

在表 8-1 的释义中，create、join、update 三个动作都使用到了配置文件（数据）：

1. channel.tx - application

channel 的创建配置文件，供 peer channel create 使用，由 configtxgen 工具根据指定的 profile 从 configtx.yaml 中读取配置数据。这里的 profile 指的是 configtx.yaml 中 Profiles 项下定义的某一个配置项。该文件主要规定了 application channel 中包含哪些组织，最终会被用作 channel 的配置原型，通过填补，作为 channel 账本的 genesis 块。

例子：

```
configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel.tx -channelID mychannel
```

上面代码指定了要生成的 application channel 的 ID 为 mychannel，并从 configtx.yaml 的 Profiles 中选取 TwoOrgsChannel 项作为 channel 的具体配置，并把生成配置数据导入到 ./channel.tx 文件中。channel.tx 中存储的是二进制格式的 Envelope，这点可以参看 common/configtx/tool/configtxgen/main.go 中的 doOutputChannelCreateTx。

2. mychannel.block - application

channel 的 genesis.block，供 join 使用，由 peer channel create 动作生成。上文中说 channel.tx 只是配置原型，在 create 的过程中，会根据 system channel 中的配置，对 channel.tx 中的数据进行详细填补，如填补 orderer 的配置（毕竟 application channel 是存在于 orderer 节点中的，不能对 orderer 的规矩一无所知）、填补 application channel 所包含的组织的详细配置。填补了这些东西，最终生成一个 block，并作为 application channel 的 genesis 块被保存入账本。要想 join 一个 application channel，就需要先获取这个 channel 的 genesis 块。

3. Org1MSPanchors.tx - application

channel 的某组织升级配置文件，供 peer channel update 使用，由 configtxgen 工具根据指定的组织 ID 从 configtx.yaml 中指定的 profile 项中生成。channel 的配置中，基本项目之一就是频道所包含的组织了，而 update 命令就是升级 channel 配置中的某个组织的配置。例子：

```
configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP
```

上面的代码指定从 configtx.yaml 的 Profiles 下的 TwoOrgsChannel 项中获取 ID 为 Org1MSP 的组织的配置数据, 用于升级频道 ID 为 mychannel 的 application channel, 并把获取生成的配置数据导入到 ./Org1MSPanchors.tx 文件中。同样, Org1MSPanchors.tx 存储的是二进制的 Envelope, 可参看 common/configtx/tool/configtxgen/main.go 中的 doOutputAnchorPeersUpdate。

8.5.3 命令

channel 的命令主要有 create、update、join3 个, 用于进行通道的一系列操作。

1. create/update

create 和 update 的操作从字面上讲一个用于创建, 一个用于升级, 而在 Fabric 中都被处理成升级操作, create 作为从无到有的升级操作, update 作为从旧到新的升级操作。在 peer/channel/create.go 和 update.go 中:

(1) 如果把 create 的执行过程分为三步, 那么 update 只是 create 的第一步而已。在 create.go 的 createCmd → create → executeCreate(cf) 中, 过程可以简单地分为:

① sendCreateChainTransaction(cf), 向 orderer 发送创建 application channel 的配置数据。

② getGenesisBlock(cf), 多次尝试从 orderer 获取生成的 application channel 的 genesis 块, 即此时 application channel 已经建立, 上一步发送的配置数据经过填补已经作为第一块 block 被保存到频道的账本中。

③ ioutil.WriteFile(file, b, 0644), 将获取的 genesis 块写入的文件中, 供 peer channel join 命令使用。在 update.go 的 update(...) 中, 完成的任务与 execute Create 中的 sendCreateChainTransaction(cf) 一致, 只不过发送的配置数据是用于更新 application channel 中的某个组织的配置数据。以下将主要以 create 的执行过程为例, 辅以 update 的不同之处进行讲述。

(2) 开始第一步。在 sendCreateChainTransaction 中, 首先根据命令行指定的 channelTxFile, 即 channel.tx 文件路径 chCrtEnv, err = createChannelFromConfigTx(channelTxFile) 读取配置数据到 chCrtEnv 中。然后通过 sanityCheckAndSignConfigTx(chCrtEnv), 检查并对 chCrtEnv 进行签名, 这样的检查不是非要不可, 因为 channel.tx 本身是 configtxgen 工具生成的, 只要工具没问题, 那这里不检查也可以。后边的步骤中还会有多次这样的 sanity 类型的检查。签名即哪个 peer 节点发起的节点信道创建, 在 chCrtEnv 中留下签名即可。最后通过 broadcastClient.Send(chCrtEnv), 向 orderer 节点发送 chCrtEnv。

(3) 在 Broadcast(...) (位于 orderer/server.go) 中的 Handle(...) (位于 s.bh.Handle(srv) -> orderer/common/broadcast/broadcast.go) 中的 msg, err := srv.Recv(), orderer 接收到第 2 点中 peer 发来的原始的配置信 msg, 接着开始从 msg 中抽取

数据。因为是配置信，因此会进入 `if chdr.Type == int32(cb.HeaderType_CONFIG_UPDATE)` 分支，在此分支中，`msg, err = bh.sm.Process(msg)` 对原始的 `msg` 进行了重新整理：过滤重复的配置，若是 `peer channel create` 的话，过滤后的配置还会被装进一个以 `system channel` 的 ID 为外衣的配置信中，形成新的配置信 `msg`。

(4) `bh.sm.Process(msg)` 最终调用的是 `orderer/configupdate/configupdate.go` 的 `Process(...)`，在此函数中，`support, ok := p.manager.GetChain(channelID)` 获取了所需要的资源，也验证了配置信所对应的频道是否存在。若存在，则说明此配置信是用于 `update` 已有频道配置的，此时进入 `if ok` 分支执行 `p.existingChannelConfig(...)`；否则，则说明此配置信是用于 `create` 新频道的，此时跳过 `if ok` 分支去执行 `p.newChannelConfig(...)`。这里 `p.existingChannelConfig(...)` 依然算是 `p.newChannelConfig(...)` 中的一部分。

(5) 在 `p.newChannelConfig(...)` 中：

①运行代码 `ctxm, err := p.manager.NewChannelConfig(envConfigUpdate)`，在 `orderer/multichain/manager.go` 的 `NewChannelConfig`，相当于验证原配置数据，并填补了一些 `system channel` 的配置。`common` 下的 `config` 和 `configtx` 生成一个内含 `application channel` 新配置的 `configManager`（这个过程相当复杂），但是 `configManager` 仅停留在当前函数中且没有返回供调用者继续使用。`envConfigUpdate` 虽然是以指针的形式传进去的，但是在 `NewChannelConfig` 配置中并没有改变 `envConfigUpdate` 中的值。而后面会介绍的 `newChannelConfigEnv, err := ctxm.ProposeConfigUpdate(envConfigUpdate)` 又直接使用原 `envConfigUpdate` 来生成 `application channel` 的频道配置信。这里就属于上文提及的类似于 `sanity` 类型的验证，因为后续步骤在真正创建 `application channel` 时也会经历 `NewChannelConfig` 和创建频道所使用的 `configManager`，所以这里相当于事前先创建一下，若有问题早发现早返回。

② `newChannelConfigEnv, err := ctxm.ProposeConfigUpdate(envConfigUpdate)`，根据原有的配置数据，利用 `configManager` 的 `ProposeConfigUpdate` 功能，通过对比配置信中读集和写集，将已有且版本相同的配置项过滤掉，生成要创建的 `application channel` 的频道配置数据 `newChannelConfigEnv`。这里的读集和写集可以生成 `channel.tx` 的函数 `doOutputChannelCreateTx`。`peer channel update` 所调用的 `existingChannelConfig(...)` 中执行的是 `support.ProposeConfigUpdate(...)`，与这里的 `ctxm.ProposeConfigUpdate(...)` 实际是一样的，都是 `configManager` 的 `ProposeConfigUpdate`，只不过 `existingChannelConfig(...)` 中使用的是已存在的 `application channel` 中的 `configManager`（包含在 `chainSupport->ledgerResources` 中）的 `ProposeConfigUpdate`。这个已存在的 `configManager` 正好也证明了后文在创建频道时还会创建频道所使用的 `configManager`。

③ `newChannelEnvConfig, err := utils.CreateSignedEnvelope(...)`，

对新生成的频道的配置信进行签名, 类型为 `HeaderType_CONFIG`。peer channel update 所调用的 `existingChannelConfig(...)` 同样执行了这一步, 且就此返回该配置信。

④ `p.proposeNewChannelToSystemChannel(...)`, 将签名过的配置信装到一个以 `system channel` 为频道 ID 的配置信中, 且类型变为 `HeaderType_ORDERER_TRANSACTION`, 然后返回新的配置信。这一点是 peer channel create 独有的, 因为创建 application channel 要使用 system channel 的 `chainSupport` 对象。

(6) 重回 `orderer/configupdate/configupdate.go` 的 `Process(...)` 中, 对原始的配置信处理之后, 通过 `support, ok := bh.sm.GetChain(chdr.ChannelId)`, 根据频道 ID 获取 `chainSupport` 对象, peer channel create 获取的是 system channel 的 `chainSupport`, peer channel update 获取的是对应 application channel 的 `chainSupport`。

(7) `_, filterErr := support.Filters().Apply(msg)`, 由上面获取的 `chainSupport` 获取频道的过滤器集合并对配置信进行 `Apply()` 过滤。只讲 peer channel create, 获取的是 system channel 的过滤器集。具体为 orderer 启动时, 在 `orderer/multichain/chainsupport.go` 的 `createSystemChainFilters(...)` 中生成。依次进行非空、大小、签名的检查后, 最后调用 `systemChainFilter` 的 `Apply()`, 即 `orderer/multichain/systemchain.go` 的 `Apply(env)`。在 `systemChainFilter` 的 `Apply(env)` 中, 抽取配置信并检查之后。

① `scf.authorizeAndInspect(configTx)` 对配置信进行整理和检查。这里需要明确一点, 从 peer 发送到 orderer 接收, 配置信均只有配置项而没有具体的配置值, 这是不完整的。而直到这一步才对配置信进行配置值的填充。在 `authorizeAndInspect(...)` 中, 我们如愿看到了 `NewChannelConfig`、`configtx.NewManagerImpl(...)` 的身影, 即根据 system channel 的配置填充配置信对应配置项的值和新建 application channel 所使用的 `configManager`。这也同时证明了上面有关 `sanity` 类型检验的说法所言非虚。

② `return filter.Accept, &systemChainCommitter{...}`, 返回 `Accept` 动作和包含了完整配置信的 `systemChainCommitter`。这里注意, 这一步主要目的有两个: 验证和填充配置信。system channel 的过滤器集合 `Apply()` 后返回执行器集合, 这是因为后文会再次生成, 在 block 写入账本之前调用执行器。

(8) `support.Enqueue(msg)`, 把配置信作为一条消息发送给 kafka (或 solo), 具体是由 `orderer/kafka/chain.go` 中 `chainImpl` 的 `Enqueue(...)` 发给 kafka 的 (又由于 `support` 是 system channel 的 `chainSupport`, 因此这里的 `chainImpl` 是与 system channel 对应的对象, 其成员 `support` 也是 system channel 的 `chainSupport`), 经过 kafka 暗盒的排序处理, 配置信被包裹在一条 `KafkaMessage_Regular` 消息中, 在 `orderer/kafka/chain.go` 的 `processMessagesToBlocks()` 中被接收, 并进入 `case *ab.KafkaMessage_Regular`: 分支, 交由 `processRegular(...)` 处理。在 `processRegular(...)` 中, 将配置信抽取出来后:

① `batches, committers, ok := support.BlockCutter().Ordered(env)`, 由配置信生成 block, 由于是配置信, 因此会单独作为一个 block, 且 `support` 是

system channel 的 chainSupport, 获取的执行器集合 committers 也是 system channel 的过滤器集合 Apply(env) 之后返回的, 即上面提及的被 “_” 省略掉的过滤器集合, 在这里又被重新生成。执行集合中只包含前文所述的 systemChainCommitter。

② for i, batch := range batches, 在循环中依次处理每批消息, 将每批消息打包成 block, 然后能过 support.WriteBlock(block, committers[i], ...) 在账本中写入 block。

③ support.WriteBlock(...) 执行的是 orderer/multichain/chainsupport.go 的 WriteBlock(...), 在这个函数中, 首先在 for _,committer := range committers 循环中依次执行 committer.Commit(), 即将执行器集合兑现。这里因为只有一个 systemChainCommitter, 因此执行的是 orderer/multichain/systemchain.go 的 Commit() -> scc.filter.cc.newChain(scc.configTx), 最终执行的是 orderer/multichain/manager.go 的 multiLedger 的 newChain(...)。在这里, 正式创建了 application channel。可以看到, 创建 application channel 的最终效果就是: 创建了频道的账本 (且写入配置信作为 genesis 块) 并在 orderer 的 multiLedger 对象成员 chains 中添加一个 chainSupport 对象并启动相应的服务。而上文提及的 peer channel update 使用到的 chainSupport 对象, 也就是这里创建的。

④还是在 WriteBlock(...) 中, 兑现执行器集合后, cs.ledger.Append(block) 也会把 application channel 的 genesis 块写入 system channel 的账本。执行至此, peer channel create 在 orderer 端的工作基本结束。

(9) 重新返回到 orderer/common/broadcast/broadcast.go 的 Handle(...) 中, 定位到 support.Enqueue(msg), 之后的 srv.Send(..._SUCCESS) 就是 orderer 节点处理完毕创建新的 application channel, 向发起 peer channel create 动作的 peer 节点返回成功的应答。这里注意一下, peer 节点接收这个应答的地方是在 broadcastClient 的 Send(...) 接口内部 (参看 peer/common/ordererclient.go 的 Send(...) 实现中的 getAck()), 也即当在 peer/channel/create.go 的 sendCreateChainTransaction 中执行完 broadcastClient.Send(chCrtEnv), 在 Send(...) 内部已经接收到来自 orderer 节点的应答信息。至此, 第一步结束。对于 peer channel update 动作来说, 至此整个动作结束。

(10) 开始第二步。重新返回到 peer/channel/create.go 的 executeCreate(cf) 中: 通过 block, err = getGenesisBlock(cf), 在一定时限内 (默认 5s, 可由 peer channel create 命令行的 -t 指定), 利用 deliver 服务每隔 200ms 向 orderer 节点指定的 application channel 索要一次序号为 0 的 block, 即上文创建的 application channel 的 genesis 块, 直到成功获取或超时。

(11) 开始第三步。通过 b, err := proto.Marshal(block) -> ioutil.WriteFile(file, b, 0644), 将第二步获取的 block 以 application channel ID+block 的格式写入文件, 供 peer channel join 动作使用。至此, 整个 peer channel create 动作执行完毕。

2. join

peer channel joinjoin 的动作是 peer 节点的本地数据和服务, 以便和对应存在于 orderer 节点中的 application channel 的数据和服务相配套。数据主要指账本, 服务主要指 gossip 等模块的服务。这里假设 peer channel create 创建的是一个 ID 是 mychannel 的 application channel, 对应生成的即为 mychannel.block 文件。在 peer/channel/join.go 中:

(1) joinCmd(cf) -> join(...) -> executeJoin(cf) 中:

① spec, err := getJoinCCSpec() 创建了一个关于 csc 的 ChaincodeSpec 格式的“说明书”, 其中的 ChaincodeInput 作为 scc 执行的输入参数, 指定了动作 csc.JoinChain、从 mychannel.block 中读取的 mychannel 的 genesis 块数据。

② invocation := &pb.ChaincodeInvocationSpec{...} -> prop, err = putils.CreateProposalFromCIS(...) -> signedProp, err = putils.GetSignedProposal(...), 包装 + 签名, 形成一个背书申请。

③ cf.EndorserClient.ProcessProposal(...), 通过背书客户端, 发起背书请求。

(2) 背书过程不再赘述。

(3) 在 configure.go 的 Invoke(stub) 中, 根据上面所述的“说明书”, 将进入 case JoinChain: 分支:

① block, err := utils.GetBlockFromBlockBytes(args[1]) 从第二个参数中获取 mychannel 的 genesis 块, 然后进行一系列的检查验证。

② joinChain(cid, block), 最后执行 join 的具体动作。

(4) 在 joinChain(cid, block) 中:

① peer.CreateChainFromBlock(block) (core/peer/peer.go), 创建 peer 节点本地的针对 mychannel 的链(账本)和 gossip 服务。在这之前, 以 chaincode、gossip 作为主题的文章中涉及和使用的账本、gossip 服务, 均是在此生成的。

② peer.InitChain(chainID) (core/peer/peer.go), 初始化 peer 节点本地的链。

③ producer.SendProducerBlockEvent(block), 创建事件服务, 此为监控服务。

(5) 在 CreateChainFromBlock(block) 中:

① l, err = ledgermgmt.CreateLedger(cb), 根据 mychannel 的 genesis 块, 创建 peer 本地专供 mychannel 使用的账本。

② createChain(cid, l, cb), 创建针对 mychannel 使用的链对象, 在这个函数中, 着重看一下 service.GetGossipService().InitializeChannel(...), 该句初始化了 peer 节点本地 gossip 模块中专供 mychannel 使用的 gossip 服务, 并在 gossip 服务与 orderer 节点之间建立了服务连接, 由此 peer 节点就可以源源不断地从 orderer 节点中的 mychannel 频道索要 block 数据块并添加到本地的用于 mychannel 的账本。

(6) 在 `InitChain(chainID)` 中：只执行了 `chainInitializer(cid)`，而 `chainInitializer` 这个函数变量是在 `peer` 节点启动起来的时候被赋值的，即在 `peer/node/start.go` 的 `serve` 中执行 `peer.Initialize(...)` 时被赋值的，这里给 `chainInitializer` 所赋的值是 `func(cid string) {scc.DeploySysCCs(cid)}`，即部署了指定频道 ID 的 system chaincode，也即初始化 `peer` 节点的 mychannel 的链，其实就是在 mychannel 链上部署的 scc。如此，`peer chaincode ...` 动作在 mychannel 上执行的过程中所使用到的 scc 也就绪了。这里所部署的针对 mychannel 的 scc 要与 `peer` 节点启动时部署的 scc (`peer/node/start.go` 的 `serve` 中执行的 `initSysCCs()`) 做区分，即如 `peer chaincode ...` 命令若是没有指定频道 ID，默认使用的是 `peer` 节点启动时部署的 scc，否则使用的是具体的针对某一频道 ID 的链上的 scc。

(7) 重回 `core/scc/csc/configure.go` 的 `joinChain(cid, block)` 中，当第 4 点执行完毕，则背书过程基本结束，开始一路返回，过程省略，可以直接定位回 `peer/channel/join.go` 的 `executeJoin(cf)` 中。在接到背书的返回结果 `proposalResp` 后，整个 `join` 动作也宣告完成。

8.6 事件机制

在 Fabric 中，采用事件的形式来通知客户端交易完成及写入区块或者智能合约事件相关信息。

8.6.1 Fabric 中 Event 相关实现

本节讲解在 Fabric 节点中如何实现事件的订阅，以及事件信息的推送。

1. events/consumer/adaptor.go

`EventAdapter` 是一个从 fabric event 服务器注册感兴趣的事件以及接收消息的一个客户端接口：

- ❑ `GetInterestedEvents`;
- ❑ `Recv`;
- ❑ `Disconnected`。

2. events/consumer/consumer.go

`EventsClient` 结构体持有 `grpc` 连接的流和用于与消费者对接的适配器 `adapter`。

- ❑ `NewEventsClient`：用于初始化一个连接某个 `peer` 的 `EventsClient`。
- ❑ `newEventsClientConnectionWithAddress`：返回一个 `grpc` 连接。
- ❑ `send`：将 Event 签名后通过 `grpc` 的 stream 发送到 event 服务端。
- ❑ `RegisterAsync`：异步注册对应的事件到 fabric event 服务器，不等待响应，调用上面的 `send` 方法。

- ❑ register: 注册感兴趣的事件到 event 中, 发送到服务端, 同步获取服务端发过来的 Event。
- ❑ UnregisterAsync: 反注册客户端感兴趣的事件, 不等待响应, 调用的上面的 send 方法。
- ❑ Recv: 从 grpc 的 stream 中接收 Event, 在客户端没有调用 Start() 方法的时候使用。
- ❑ processEvents: Start 方法中重新创建一个 goroutine 调用的方法, 用于接收服务端发过来的 Event。
- ❑ Start: 建立和事件中心的连接, 获取客户端感兴趣的事件, 同步注册到服务端, 并启动一个新的 goroutine, 调用 processEvents 处理服务端发来的 Event 信息。
- ❑ Stop: 终端和事件中心的连接。
- ❑ getCreatorFromLocalMSP: 获取 signer 的字节数组。

这个文件主要在 block-listener.go 中有使用, 每个 SDK 中对应都有类似的实现, 最为相似的应该是 fabric-go-sdk。

consumer.go 使用方法:

- (1) 根据自身需求实现 EventAdapter。
- (2) 使用 NewEventsClient 生成 EventsClient 实例。
- (3) 调用 EventsClient 的 Start 方法: 建立与 eventhub 服务器的连接, 同步注册一开始设置的感兴趣的事件类型。启动一个 goroutine 执行 processEvents 方法, 该方法是阻塞型的, 用于不断接收来自 eventhub 的消息, 调用客户端实现的 EventAdapter 的 Recv 方法。

具体示例请参考 block-listener.go 文件。

8.6.2 events/producer

本节主要介绍 events/producer 包下的文件。

1. producer.go

EventsServer 结构体: 实现 grpc 的 Chat 的结构体。

2. events.go

handlerList: 事件处理器接口。

```
type handlerList interface {
    add(ie *pb.Interest, h *handler) (bool, error)
    del(ie *pb.Interest, h *handler) (bool, error)
    foreach(ie *pb.Event, action func(h *handler))
}
```

该事件分为:

- ❑ genericHandlerList: 普通处理器列表。

```
type genericHandlerList struct {
    sync.RWMutex
    handlers map[*handler]bool
}
```

handlers 这种表示方法是 go 语言里实现 set 的常用方法。

❑ chaincodeHandlerList: 链码处理器列表。

```
type chaincodeHandlerList struct {
    sync.RWMutex
    handlers map[string]map[string]map[*handler]bool
}
```

handlers 表示对应的 chaincodeId 下的 eventName 的 map[*handler]。bool 是 chaincodeHandlerList 中的特殊数学结构, 因此其分为 2 个结构体。

genericHandlerList 的三个实现方法:

❑ add: 将 handler 添加到 handlerList 中。

❑ del: 将 handler 从 handlerList 中移除。

❑ foreach: 遍历 handlerList, 执行 action。

chaincodeHandlerList 的三个实现方法: 其中 ie *pb.Interest 这个参数在这个实现中没有用到:

❑ add: 操作加了写锁, 判断相关参数是否为空, 将对应的 handler 添加到对应的 chaincodeId 下对应的 eventName。

❑ del: 操作加了写锁, 判断相关参数是否为空, 将对应的 chaincodeId 的 eventName 的 handler 移除。

❑ foreach: 操作加了写锁, 获取指定 chaincodeId 下的 EventName 下的所有 handler, 执行 action, 此方法中还有一段代码 (第 114~159 行), 永远不会执行估计。

3. handler.go

❑ handler: 用于向 eventHub 注册客户端感兴趣的事件以及仅注册相应的事件。

❑ newEventHandler: 创建一个 handler。

4. register_internal_events.go

❑ getMessageType: 获取消息类型。

❑ addInternalEventTypes: 添加相应的事件类型。

5. examples/events/block-listener/block-listener.go

为 adapter 结构体实现 consumer.EventAdapter 接口的三个方法: -GetInterestedEvents、-Recv 和 -Disconnected。这 3 个方法是客户端自定义逻辑的实现。

createEventClient: 调用 NewEventsClient 生成 EventsClient 实例, 并调用 Start 方法。

main 方法 --->createEventClient, 没有事件返回时进行阻塞; 有事件返回时则解析事件, 并进行相应的输出。

8.6.3 Go SDK 中 Event 相关实现

本节将介绍在 Go SDK 中如何实现客户端订阅相关的事件信息。

1. EventHub

EventHub 主要用于注册和反注册相应的事件：

- ❑ SetPeerAddr：设置事件注册地址，用于指明在哪个节点上注册该事件。
- ❑ IsConnected：判断客户端与事件服务端的 grpc 是否保持连接状态。
- ❑ Connect：和事件服务器建立连接。
- ❑ Disconnect：和事件服务器断开连接。
- ❑ RegisterChaincodeEvent：注册一个 ChaincodeEvent 事件。
- ❑ UnregisterChaincodeEvent：取消注册一个 ChaincodeEvent 事件。
- ❑ RegisterTxEvent：注册一个交易类型事件。
- ❑ UnregisterTxEvent：取消注册一个交易类型事件。
- ❑ RegisterBlockEvent：注册一个 Block 类型的事件。
- ❑ UnregisterBlockEvent：取消注册一个 Block 类型的事件。

2. EventsClient

EventsClient 主要用于管理事件的异步发送及连接的管理：

- ❑ RegisterAsync：异步注册需要注册的事件。
- ❑ UnregisterAsync：异步取消注册相应的事件。

3. EventHubExt

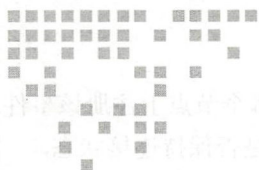
EventHubExt 用于对 SDK 添加额外的扩展功能。

4. ChainCodeCBE

在 EventHub 内部使用，用户持有 chaincode event 的注册回调。

5. ChaincodeEvent

ChaincodeEvent 相关信息的封装。



Chapter 9 第9章

BCCSP 加密服务提供者的设计与实现

BCCSP 为加密服务提供者，为 Fabric 网络信息传输提供了密码学的保障。

9.1 密码学相关知识介绍

9.1.1 安全基础

当在互联网上交换数据时，被传输的数据会经过各种各样的设备和网络后才能到达目的地，正是因为如此，安全技术在互联网时代才显得如此重要。

数据在网络上传输的过程中，存在着 4 个比较显著的问题：

(1) 当 A 想要向 B 发送消息时，可能存在第三方 X 来截获消息的内容，这种问题称为 interception。

(2) 即使 A 本来要将消息发送给 B，也存在 X 伪装成 B 来接收消息的可能性；同样的，也存在 B 相信接收到的消息来自于 A，但实际上是 X 伪装成 A 来发送这条消息，这种问题被称为 spoofing。

(3) 假设 A 向 B 发送的消息已经完成了，也存在在消息发送过程中，X 篡改消息内容的可能性，这种问题被称为 falsification。除了有意篡改消息以外，也存在消息在发送的过程中因为传输设备出现异常而出错的可能性。

(4) 假设 B 已经正确地接受到了 A 发送的消息，但是发送方 A 出于自己的私心考虑，否认了这条消息是由他自己发出的，当出现这种情况时，互联网将无法成为商业交易或商业合同签署合格的媒介，这种问题被称为 repudiation。

当然，上述 4 个问题并不仅存在于人与人之间的信息交换，同时存在于我们日常上网

的过程中。下面我们来简单介绍一下解决上述问题的安全技术。

为了解决消息被截获的问题，我们可以使用加密技术；解决第二个问题的方法可以是采用消息验证码或者数字签名技术。解决第三个问题也可以使用消息验证码或数字签名技术。数字签名技术还可以用于解决第四个问题。

其实数字签名本身也存在问题，就是它本身无法验证签名使用的公钥的拥有者的真实性，所以就出现了数字证书，以此来解决数字签名本身的问题。

9.1.2 加密基础

为什么在现代的互联网世界中我们需要加密技术呢？例如 A 需要向 B 发送消息，在消息到达 B 之前会经过各式各样的设备和网络，所以有很大的可能消息会被恶意第三方所截获，因此出于这种原因，在发送消息之前对消息加密就显得非常有必要了。

被加密的数据被称为密文，消息将会以密文的形式发送给 B，B 收到 A 发送过来的密文后，将其解密后就可以获得原来的消息。将加密的数据还原成数据的原始状态的过程被称为解密。当你将数据以加密的形式发送出去时，就可以不必担心恶意的第三方截获到你的数据了，正因为如此，加密技术在现代社会中才显得如此重要。

下面我们来仔细了解一下在加密过程中涉及的操作。我们知道，在计算机中，无论数据的类型是什么，计算机都将它们视为一系列二进制数据，也就是一长串的 0 和 1 的组合。所以，尽管存在各种各样的数据格式，例如文本文件、音乐文件和影像文件。在计算机中，所有的数据都是以二进制数据的形式进行管理的。

同样，在加密的过程中，我们也可以以计算机管理数据的方式来思考我们的数据。数据就是一系列有意义的数字，虽然密文同样也是一串数字，但是由于它们的随机性，计算机并不能正确地解释它们，加密就是通过计算来将原来对于计算机来说有特定含义的数据转换成计算机无法解释的数据的过程。在加密的过程中，存在一个用于执行计算的密钥。这个密钥同样也是由数字组成的。换一句话说，加密就是通过密钥来进行数值计算，将原本可读的数据转化为不可读数据的过程。反过来，解密就是通过密钥来进行数值计算，将不可读的数据转化为可读数据的过程。

我们通过一个“异或”的例子来理解加密和解密的过程，XOR(异或)的结果是根据真值表决定的，我们有： $0 \text{ XOR } 0 = 0$ ； $1 \text{ XOR } 1 = 0$ ； $0 \text{ XOR } 1 = 1$ ； $1 \text{ XOR } 0 = 1$ 。

“异或”操作的一个特点就是 $A \text{ XOR } B = C$ ， $B \text{ XOR } C = A$ ，这意味着如果 C 是 A 和 B “异或”的结果，那么你就可以通过 C 与 A 和 B 中任意一个值进行“异或”操作来获得另外一个数字。将这个知识点用于加密和解密的过程中，我们可以将私钥和数据进行“异或”操作得到密文，同理，将生成的密文与私钥再进行一次异或操作，我们就可以还原得到原来的明文。

从上述过程中我们可以看出，相同的私钥被用于“异或”加密或解密数据的过程中。

9.1.3 哈希函数

一个哈希函数可以将不同长度的数据映射为定长的一个值。我们可以形象地将哈希函数理解为榨汁机，我们将原始材料（我们的数据）放进榨汁机（哈希函数）得到饮料（哈希值）。我们可以将原材料转化为饮料，但是反过来饮料并不能通过一定的操作转化为原材料。类比我们的哈希函数，我们可以将原始数据通过哈希操作转化为哈希值，但是我们不可以使用哈希值得到我们的原始数据。也就是说，哈希是一个不可逆的过程。哈希值其实就是一个数值，但是我们常常使用十六进制来表示。

哈希函数的特点如下：第一个特点就是通过哈希函数得到的输出长度都是固定的。长度固定是针对特定的哈希函数而言的，例如 SHA-1 哈希函数得到的哈希值的长度都是 20 字节，这就意味着，即使你的输入数据非常大，但是通过 SHA-1 哈希之后，你得到的输出依然是固定的 20 字节。类似的，不管你的输入数据有多小，得到的输出依旧是 20 字节。哈希函数的第二个特点就是在哈希函数相同的情况下，相同的输入永远得到的是相同的输出。第三个特点也是非常重要的一个特点，就是你输入两个类似的数据，即使两个数据只相差 1bit，得到的输出相差也会非常大。哈希值不会因为你的输入数据相似而得到相似的输出。第四个特点就是可能会存在输入的数据完全不同，依旧得到两个完全相同的哈希值的情况。这种情况出现的概率非常小，这种情况被称为哈希碰撞。第五个特点就是哈希函数具有单向性或者说是不可逆性，你几乎不可能通过哈希值得到原始数据，这也是哈希函数与加密之间一个非常重要的区别。哈希函数的最后一个特点就是计算哈希值的过程可以非常简单。

我们可以使用 MD4、MD5、SHA-0、SHA-1、SHA-2 之类的哈希函数，现实生活中最常用的是 SHA-2。

9.1.4 共享密钥加密

共享密钥加密在加密和解密的过程中使用相同的私钥。

假设 A 想要通过互联网将一段消息发送给 B，在发送给 B 的过程中消息经过了很多不同的网络和设备。但是，如果 A 直接将消息以明文的形式发给 B，则恶意第三方就非常容易截获该消息。正是因为这样，所以在传输消息的过程中通常需要经过加密处理。利用私钥，将明文加密为密文再进行发送。B 利用相同的私钥将接收到的密文解密得到原文。所以，经过加密处理后，即使恶意第三方截取了消息，在不知道密钥的情况下，他也很难知道原文是什么。

利用相同密钥进行加密和解密是共享密钥加密的一个特点，常见的共享密钥加密方法有凯撒加密、AES、DES、OTP 等，其中 AES 使用最为广泛。

当然，共享密钥加密本身存在着一个致命的缺陷：假设 B 收到了 A 发送的密文，在发送的过程中，密文也可能被恶意第三方 X 所截获。现在存在的问题是，由于 A 和 B 可能从来没有直接接触过，B 可能并不知道 A 用于加密的密钥，所以在这种情况下，A 就需要某

种机制把密钥传输给 B，就像 A 向 B 传输密文一样，A 同样需要在互联网上向 B 传输密钥。B 利用接收到的来自 A 的密钥就可以对密文进行解密。既然是在互联网上传播，恶意第三方 X 同样有机会截获到密钥，这样一来，X 就可以用截获到的密钥对密文进行解密。

在上述过程中，我们可以清楚地看到密钥本身的传输存在问题，所以，A 可能会想通过对密钥本身进行加密后再传输，对于计算机来说，一个密钥仅是一段数据，所以我们可以使用新的密钥对老的密钥进行加密并生成密文，再对密文进行传输。但是我们现在又需要一个机制来传输新的密钥，这样就会陷入一个循环，总结一下，共享密钥加密系统缺乏一个安全传输密钥的机制，这个问题被称为密钥分发问题。解决密钥分发问题有两种方法：使用 Diffie-Hellman 密钥交换这样的交换协议；使用公钥加密算法。

9.1.5 公钥加密

公钥加密与共享密钥加密最大的区别在于公钥加密在加密和解密时使用的是不同的密钥，用于加密的密钥被称为公钥（public key），用于解密的密钥被称为私钥（private key）。除了使用不同的密钥之外，公钥加密在加密和解密时比共享密钥加密需要花费的时间更长。常用的公钥加密有 RSA 加密和 ECC 加密（椭圆曲线加密）。

下面我们来了解一下在公钥加密的情况下数据是如何在双方之间交换的。

假设 A 想要给 B 发送一段消息，首先，接收者 B 会生成一对密钥，分别是一个公钥和一个私钥，公钥部分将由 B 发送给 A，A 使用从 B 发送过来的公钥将消息加密，并将加密后的密文发送给 B。B 利用只有自己拥有的私钥将从 A 发送过来的密文进行解密，从而得到原始可读的消息。由于只有公钥和秘文在网上传输，而且只通过公钥并不能解密密文秘文，所以即使恶意的第三方截获了这两个信息，他也并不能解密密文获得原始消息。所以，公钥加密很好地解决了共享加密中私钥分发的安全问题。

使用公钥加密的另一个好处就是，在一个不确定人数的组织中交换数据非常方便。举一个例子，假设 B 已经准备好一对密钥，即一个公钥和与公钥相对应的私钥。因为公布自己的公钥并不会有什么影响，所以，B 将自己的公钥公布在互联网上，另一方面，由于私钥是绝对不能让他人知晓的，所以私钥需要牢牢地守护。现在有多个个体需要向 B 发送数据，首先这些个体需要获取由 B 分享的公钥，然后利用公钥将需要发送的数据进行加密，最后将加密后的数据发送给 B。B 利用自己的私钥分别将收到的密文解密获得原来的数据。所以，在存在多个个体时，我们并不用准备多个公钥，还有一点就是，由于只有消息的接收者才具有用于解密的私钥，所以公钥加密的安全性非常高。

然而在存在诸多优点的同时，公钥加密也存在着两个严重的问题。

（1）公钥加密所需要的时间相对较长，正是因为这样，公钥加密并不适用于交换大量数据的情景。为了解决这个问题，出现了“混合加密”这样的解决方案。

（2）公钥本身的可靠性存在问题。当 B 生成公钥和私钥时，为了方便起见，将 B 生成的公钥记为 PB，将私钥记为 SB，假设存在着恶意的第三方 X，想要截获 A 发向 B 的数据，

同时也生成了一对公私钥，分别记为 PX 和 SX。当 B 向 A 发送自己的公钥 PB 时，X 隐蔽地将 B 的公钥 PB 替换为自己的公钥 PX，并将 PX 发送给 A。由于通过公钥并不能发现公钥的产生者是谁，所以 A 并不能发现 B 的公钥有没有被第三方调包。接着 A 将自己的数据用公钥 PX 进行加密，当 A 将加密后的消息发送给 B 时，X 截获了秘文，由于该秘文是用 X 的公钥 PX 加密的，X 可以用自己的私钥 SX 来对秘文进行解密，这样 X 成功截获了本来该发送给 B 的消息。接着，X 将消息用 B 的公钥 PB 加密后发送给 B。由于过程中产生的密文是用 B 的公钥加密的，所以 B 可以利用自己的私钥 SB 对密文进行解密。正是因为 B 可以正确地对消息进行解密，A 和 B 都无法知道消息已经被第三方截获过了。

上述通过替换公钥来截获消息的攻击方式被称为中间人攻击。这种问题的关键就在于 A 无法验证自己获取的公钥到底是来自 B 还是来自恶意的第三方 X。为了解决中间人攻击问题，出现了数字证书。

9.1.6 混合加密

如果使用共享密钥加密，我们会遇到密钥分发问题——如何安全地分发密钥？另一方面，如果使用公钥加密，漫长的加密解密过程又降低了信息传输的效率，混合加密的出现就整合了两者优点，规避了两者的缺点。

混合加密利用共享密钥加密加解密的高效性的同时，使用公钥加密来保障共享密钥的安全性。

假设 A 要向 B 发送消息，可以利用共享加密方法进行高效加密操作，因为用于加密的密钥也同样被用于解密，所以 A 需要把密钥安全地发送给 B。其实密钥也是数据的一种形式，通过公钥加密，我们可以安全地将密钥发送给 B。接收者 B 自己生成一对密钥，即一个公钥和一个私钥，B 将公钥发送给 A，A 利用 B 的公钥对对称密钥进行对称加密处理，加密后的密钥再发送给 B，B 利用自己的私钥对加密后的密钥进行解密处理，这样 A 就利用公钥加密将密钥安全地发送给 B。现在剩下唯一要做的就是将经过对称密钥加密后的密文发送给 B，由于共享密钥加密解密速度快，B 就可以准确快速接收到原始数据。

上述过程我们可以清楚地看到混合加密整合了公钥加密的高安全性和共享密钥加密解密的快速性，混合加密已经被应用到 SSL 中，用于互联网上安全交换数据。

9.1.7 消息验证码

消息验证码实现了两个功能：authentication 和 falsification detection。

首先，我们来认识一下为什么消息验证码有存在的必要。假设 A 需要从 B 那里购置一件商品，A 向 B 发送自己要购买的商品编号 xyz，然后 A 将该商品编号加密，假设加密时使用了共享密钥，A 先使用某种安全的方法将共享密钥加密方法使用的密钥传递给 B，这种密钥交换方法可以是公钥加密还可以是 Diffie-Hellman 密钥交换手段。A 使用共享密钥对需要发送的消息进行加密，这里的消息就是我们需要发送的商品编号，加密后得到的密文被

发送给 B, B 得到密文后使用共享密钥进行解密后得到明文, 即 A 发送的商品编号。这个传输过程看起来好像并没有什么问题, 但是现实生活中可能发生这样的事情: 当 A 向 B 发送经过加密后的密文时, 恶意第三方 X 在传输过程中将密文篡改。当 B 收到被篡改后的密文并解密后, 得到的商品编号却变成了“123”, 而 B 并不知道商品编号已经被篡改过了, B 将商品“123”而不是“xyz”发送给 A。因为加密无非就是对原有数据进行数值计算, 所以, 解密的过程也可以在被篡改过的数据上进行, 但是如果原消息比较长, 而经过篡改后的密文解密后变成了一些没有实际意义的消息, 那么消息的接收方可能会察觉到消息已经被篡改过了。但若是一些人类无法直接识别的消息, 例如上述的商品编号, 那么接收方就很难察觉出消息被篡改过了。所以, 为了检测消息是否已经被恶意第三方篡改过, 除了加密之外再增加一些手段就变得非常有必要了。

消息验证码就是为了检测消息有无被篡改而出现的一种解决方案。下面我们来仔细看一看消息验证码是如何工作的。当 A 将加密后的密文发送给 B 之前, A 同时生成一个用于消息验证码的密钥, 然后 A 将该密钥以安全的方法发送给 B, 然后 A 利用该密钥和密文以特定算法生成一个值, 这个由密钥和密文组合生成的值就被称为消息验证码, 通常被简称为 MAC。MAC 值可以简单地理解为密钥和密文组合起来的哈希值。我们可以使用各种各样的方法来生成 MAC 值, 例如 HMAC、OMAC、CMAC 等, 现实生活中 HMAC 使用最为广泛。生成 MAC 值以后, A 连同密文和 MAC 值一同发送给 B, B 接收到以后, 需要根据 MAC 值检测密文有无被第三方篡改。B 和 A 一样, 根据密文和密钥来重新生成 MAC 值, 这样根据 A 发送过来的 MAC 和 B 自己重新生成的 MAC 进行比较, 如果两者相同则说明密文并没有被他人篡改。接下来 B 仅需将密文解密得到原文。

那么如果在 A 将密文发送给 B 的过程中, 恶意第三方 X 将密文篡改会怎么样? 假设在 A 将密文和 MAC 值发送给 B 的过程中, X 篡改了密文, 然而 B 收到以后重新计算 MAC 值, B 会发现自己计算的 MAC 值和 A 发送的 MAC 值并不相同, 这样 B 就可以判断密文、MAC 值或者两者都已经被篡改过了。在这种情况下, B 可以选择将密文抛弃然后再重新请求 A 再次发送相同的密文。是否存在 X 将密文和 MAC 值同时改变从而让 B 重新计算出来的 MAC 值和被传输的 MAC 值一致的可能呢? 由于 X 不知道用于生成 MAC 值的密钥, 所以即使 X 能篡改传输中的消息, X 也无法做到让两者的 MAC 值一致, 所以 B 在重新计算 MAC 值时发现与被篡改后消息一起携带的 MAC 值不一样时, B 就可以清楚地知道该消息已经被别人做过手脚了。

经过上面的解释, 我们可以清楚地看到, 使用消息验证码或者 MAC 可以避免消息在传输的过程中被他人篡改。然而, 消息验证码同时也存在着一些缺陷。由于 A 和 B 共享一对用于加密明文的密钥和用于生成 MAC 值的密钥, 如果 A 可以加密消息和生成 MAC 值的话, 那么 B 也可以还原这样的操作, B 同样也可以加密消息和生成 MAC 值。换句话说, 你并不能区分原文是由 A 还是 B 生成的, 那么假设 A 是恶意的, 在 A 将消息发送给 B 后, A 可以声称该消息是由 B 伪造生成的, 这样 A 就否认了该消息是由他自己发送的。为了防止

这种现象出现，我们需要采取另外一种技术，也就是下面会讲的数字签名技术。

9.1.8 数字签名

数字签名可以保证消息的不可否认性。数字签名除了完成消息验证码实现的 authentication 和 falsification 检测的功能之外，还实现了让消息的真实发送者无法否认这条消息是由他发送出的功能，这就是不可否认性。

在了解数字签名的工作原理之前，我们先回顾一下消息验证码。使用消息验证码的系统中，为了验证消息的发送者是否为真实的密钥拥有者，会将 MAC 值附着到要发送的消息上。为了方便起见，这里的消息以明文形式传输，A 先以某种安全的方法将用于生成 MAC 的密钥发送给 B，然后再将明文和 MAC 值发送给 B，B 根据收到的明文和密钥重新生成 MAC 值，然后再确认两者的 MAC 值是否相同。如果确认成功，那么就说明 A 是真实的发送者并且发送的明文没有经过他人篡改。然而由于消息验证码采用的是共享的密钥，依旧存在拥有该密码的组织伪装成为消息发送者的可能性，比如说，当 A 把一条消息发送给 B 之后，B 却声称自己才是这条消息的创建者。还有一点就是，由于采用的是共享密钥来生成 MAC 值，那么当 A 与除了 B 之外的其他个体进行消息交流时，则需要生成一个新的密钥。

为了解决消息验证码内在的问题，我们引入数字签名技术，数字签名技术不采用 MAC，而是采用只有消息的真实发送者才能产生相关数据的技术，这里产生的数据就被称为数字签名。例如 A 要发送一段消息“abc”，A 会生成一段只能由 A 生成的签名，所以，当 A 发送的带有原消息和签名的数据被接收方接收到时，接收方就可以保证发送方一定是 A。消息的接收方 B，可以判断数字签名来自发送者 A，但是他自己并不能生成这些数字签名。与消息验证码相对比，由于数字签名并没有使用共享密钥，所以 A 可以向不同的组织或个体发送相同的数字签名。我们知道用于生成 MAC 值的密钥是两方共享的，而当生成数字签名时，我们使用了和公钥加密系统类似的一个过程。

我们在这里简单地回顾一下公钥加密的过程：首先如果 A 有消息要发送给 B，接收方 B 会事先生成一对密钥，分别是公钥 P 和私钥 S，B 将公钥 P 传递给 A，A 使用 B 的公钥去加密自己的数据，然后再将加密后的密文发送给 B。B 利用只有自己才拥有的私钥 S 对密文进行解密，整个过程结束。所以在公钥加密中，公钥被用于加密而私钥被用于解密，正是因为如此，任何人都可以使用公钥进行加密，但因为只有 B 才拥有私钥，故只有 B 才能进行解密。但如果我们将公钥加密的操作反向进行会怎么样呢？将私钥用于加密，而将公钥用于解密，在这种情况下，假设只有 A 才有私钥，只有他们才能加密数据，而加密生成的数据可以被任何持有对应公钥的人进行解密，但这样的加密手段没有任何意义。如果换一个角度去看的话，我们就可以理解为密文的生成者必定是持有私钥的 A 而不会其他个体。

下面我们来仔细了解一下数字签名的整个流程。首先 A 需要准备发送的数据及一对公私钥。注意，由消息的发送者准备公私钥是与公钥加密不同的地方之一。在公钥加密中，公私钥是由消息的接收方所准备的。A 将自己的公钥发送给 B，然后 A 利用自己的私钥对

消息进行签名, 签名结果就被用来当作数字签名, A 将消息和数字签名一同发送给 B, B 利用 A 的公钥对数字签名进行验证。如果验证的消息与收到的消息一致, 那么整个交换的过程就结束了。

用 A 的私钥生成的数字签名, 可以用 A 的公钥进行验证, 但是该签名却只能由 A 生成, 所以我们可以确认 A 才是消息的发送者并且发送的消息没有被别人篡改过。而且, 由于 A 的签名不能有除了 A 以外的组织 (例如 B) 生成, 因为 B 只有公钥, 所以数字签名同时也实现了不可否认性这一功能。然而, 和公钥加密类似, 加密和解密与签名和验证都需要花费很多时间, 所以, 我们不直接对消息进行签名。为了减少计算所需要的时间, 我们首先生成与消息对应的哈希值, 接着对哈希值进行签名操作再得到数字签名, 接着将原消息和利用哈希值得到的数字签名发送给 B。同样, B 利用收到的消息计算哈希值, 并利用公钥对收到的数字签名进行解密操作得到另一个哈希值。如果这两个哈希值相同, 那么说明整个过程没有出错, 利用数字签名进行交换的过程也算结束了。

数字签名技术提供了 authentication、falsification detection 和 repudiation prevention 功能, 但是它本身依旧存在着一个问题, 消息和数字签名的接收方 B 会无条件地认为 A 一定是消息和数字签名的发送方。但是事实上, 很可能存在恶意第三方 X 伪装成 A 进行数据交换, 存在这个问题的根本原因就在于数字签名和公钥加密都使用了一个公钥和一个私钥, 我们并不能确定这个公钥到底属于谁。因为公钥中并没有包含它的归属方的信息, 所以 B 使用的公钥很可能并不来自 A, 而是来自恶意第三方 X 的公钥。这个问题我们可以用“数字证书”这一技术来解决。

9.1.9 数字证书

公钥加密体系和数字签名存在的共同的问题就是无法确定公钥的归属者。所以, 当 A 需要将自己的公钥发送给 B 时, 恶意第三方 X 可以在不让 A 和 B 发现的情况下将 A 的公钥替换为自己的公钥。数字证书就是为了解决“无法确定公钥归属者”这一问题而出现的解决方案。

下面我们来了解一下数字证书是如何工作的。假设组织 A 拥有一对公私钥, 分别记为 PA 和 SA, 并且决定将 PA 发送给组织 B。组织 A 需要向 CA(certification authority) 发起请求, 让 CA 颁布一个证明公钥 PA 属于组织 A 的证书。那什么是 CA 呢? CA 就是用于管理数字证书的一个组织, 一般来说, 任何人都可以称为 CA, 但是最好还是相信由政府或者大型公司这些信用度高的组织担任的 CA。同样, 每个 CA 自身也拥有一对公私钥, 组织 A 向 CA 提供自己的个人信息, 例如公钥 PA、电子邮箱等。当 CA 收到由组织 A 提供的信息时, CA 利用自己私钥对组织 A 提供的数据进行签名, 得到数字签名, 然后将生成的数字签名和 A 提供的消息一起生成一个文件并发送给 A, 这个文件就变成了 A 的数字证书。

从现在开始, A 就不向 B 直接发送自己的公钥 PA 了, 取而代之的, A 向 B 发送来自 CA 生成的数字证书。在那之后, B 向 CA 索取它的公钥, 并利用 CA 的公钥验证 A 发送的

数字证书的真实性。数字证书中的数字签名仅可以用产生该数字证书的 CA 的公钥来验证，换句话说，如果验证不出问题，那就可以说明该数字证书确实是由该 CA 颁布的，然后就可以提取数字证书中 A 的公钥 PA 了。经过这个过程，组织 A 向组织 B 传输公钥的过程就算完成了。

让我们看看这种公钥传递方式是否存在问题。假设一个冒充 A 的恶意第三方 X 试图传递自己的公钥，但是 B 没有理由相信一个没有作为数字证书发送的公钥。那么当 X 冒充 A 在认证机构注册公钥时会发生什么？在这种情况下，X 无法访问 A 的电子邮件账户，因此 X 无法获得发行证书。X 只能创建使用 X 的电子邮件地址的证书。因此，他们无法获得属于 A 的证书。通过使用数字证书系统，可以验证公钥的所有者。早些时候我们说 B 收到了证书颁发机构的公钥，但是现在存在一个问题：B 收到的公钥（PC）是否真的是由认证机构创建的？因为没有办法确认是谁创建了公钥，它可能是由 X 伪装为认证机构创建的。换句话说，我们在公钥加密中发现的同样的问题也出现在这里了。实际上，认证机构的这个公钥（PC）也作为数字证书被交付，并且“签名”认证机构的数字证书的一方是更高等级的认证机构。认证机构形成了一个树状结构，高级权威机构为较低级别的机构创建数字签名。

下面来了解这个认证机构的树状结构是如何运作的。比方说，我们有一个社会广泛信任的认证机构 Y，即使新公司 G 想要作为认证机构开始服务，它在社会上也是没有信誉的。因此，G 公司需要拥有 Y 公司颁发的数字证书。当然，Y 公司会检查以确保 G 公司能够充分履行认证机构的职责。在这之后，G 公司就可以宣称自己是一家赢得 Y 公司信任的公司。通过这样做，大组织可以确保小型组织拥有信任度，从而形成有组织的树状结构。那么，谁在认证机构树的顶部？最高级别的证书颁发机构被称为根证书颁发机构（root CA），这类机构自己证明其自身的有效性。另外，如果根证书颁发机构本身不是一个值得信赖的组织，那么它颁发的证书也不会被使用。因此，现实中相当一部分 CA 是已经具有社会公信力的组织，如大公司和政府机构。

9.2 BCCSP 概要

BCCSP（BlockChain Cryptographic Service Provider）即区块链加密服务提供者，这个模块主要提供了区块链中需要使用的密码学标准和算法，是整个项目中非常重要的一个模块，是保证联盟链安全的基石。

9.2.1 BCCSP 简介

BCCSP 在设计的时候考虑了一个非常重要的特点，就是可插拔性，即在不改变 Fabric 核心代码的前提下，可以提供不同密码学方法的实现。

```

type BCCSP interface {

    KeyGen(opts KeyGenOpts) (k Key, err error)
    KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)
    KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)
    GetKey(ski []byte) (k Key, err error)

    Hash(msg []byte, opts HashOpts) (hash []byte, err error)
    GetHash(opts HashOpts) (h hash.Hash, err error)

    Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)
    Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)

    Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error)
    Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error)
}

```

在上述 BCCSP 的接口中，可以看出这部分主要提供了 4 个基本功能模块：

- ❑ 对密钥生命周期的管理；
- ❑ 对消息进行哈希处理；
- ❑ 对哈希生成的摘要进行签名和验证；
- ❑ 对消息进行加密（对称加密或非对称加密）。

因为接口 BCCSP 仅提供了一个通用的方法群，它并没有告诉我们具体应该使用哪一种签名方法或者加密方法。在整个 BCCSP 模块中提供了很多可供定制的选项，这一点也体现了整个模块在 hyperledger fabric 项目中的可插拔性。在 `opts.go` 中我们在常数定义部分可以看到整个模块大致实现了 SHA 系列的哈希算法、ECDSA 和 RSA 系列的签名算法、AES 系列的对称加密算法及相应系列中不同安全级别的算法等。

```

const (
    ECDSA = "ECDSA"
    ECDSAP256 = "ECDSAP256"
    ECDSAP384 = "ECDSAP384"
    ECDSARERand = "ECDSA_RERAND"

    RSA = "RSA"
    RSA1024 = "RSA1024"
    RSA2048 = "RSA2048"
    RSA3072 = "RSA3072"
    RSA4096 = "RSA4096"

    AES = "AES"
    AES128 = "AES128"
    AES192 = "AES192"
    AES256 = "AES256"

    HMAC = "HMAC"

```

```

HMACTruncated256 = "HMAC_TRUNCATED_256"

SHA = "SHA"
SHA2 = "SHA2"
SHA3 = "SHA3"
SHA256 = "SHA256"
SHA384 = "SHA384"
SHA3_256 = "SHA3_256"
SHA3_384 = "SHA3_384"

X509Certificate = "X509Certificate"
)

```

在对密钥的抽象和管理方面，不论是对称密钥还是非对称密钥，BCCSP 都统一使用 Key 这个接口来表示。

```

type Key interface {
    Bytes() ([]byte, error)
    SKI() []byte
    Symmetric() bool
    Private() bool
    PublicKey() (Key, error)
}

```

对于对称密钥来说，Symmetric() 方法返回 true，PublicKey() 方法返回一个 nil 值和 false；而对于非对称密钥来说，Symmetric() 方法则返回 false，PublicKey() 方法返回与该私钥对应的公钥值。

```

type KeyStore interface {
    ReadOnly() bool
    GetKey(ski []byte) (k Key, err error)
    StoreKey(k Key) (err error)
}

```

除了对使用密钥进行一系列的加密操作以外，密钥本身的存储机制也同样非常重要，因为机器本身存在宕机的可能性。如果所有密钥仅存在在内存中，一旦机器出现问题，将出现不可逆转的损失，所以一个可靠的密钥存储机制非常有必要。

KeyStore 代表了密钥的存储系统，在非只读的情况下，它支持两个简单的操作，即存储密钥和通过 ski 取回密钥。当在只读限制条件下，StoreKey 方法将返回 error。

因为在联盟链中大量地使用了签名算法来保证信息的真实性，所以深入了解一下签名算法尤其是 ECDSA 的原理对理解整个签名过程还是非常有增益的。

9.2.2 陷阱函数

在理解 ECDSA 之前，首先要理解签名的真实性来源与用于签名的私钥无法被伪造的特性。如果私钥可以轻易被别人获取或者破解，那利用算法得到的签名将毫无意义。所以，

私钥无法被破解（即破解的难度非常大）是整个密码学的前提条件。

利用 ECDSA 进行签名或验证需要有两样东西：私钥和公钥。私钥和公钥的关系可以简单地用陷门函数来描述。假设私钥是一个随机数 k ， P 为椭圆曲线上已知的一个点，则通过“点乘法”得到 $R=k \cdot P$ 。即使你知道 R 和 P ，你也无法通过这两个点来获取 k 值；而假如预先知道了 k 值，则我们可以非常轻松地得到 R 值。

总结一下，陷门函数的精髓就是正向演算非常简单，而逆向推导难度非常大，这就是 ECDSA 算法安全性的基础。

9.2.3 为什么要使用 ECDSA

举一个非常简单的例子，比如你将一份文件传到互联网上供他人下载，但你需要保证别人下载到的文件是真实的，所以你根据文件计算出相应的哈希值和文件放在一起，希望别人同样计算哈希值与他们下载的文件相对比。但这存在一个比较严重的问题：如果第三方可以篡改你上传的文件，那他们一定也有办法篡改相应的哈希值，所以哈希值的方法并不可靠。现在我们需要的是别人无法伪造的算法，即我们使用他人无法获取的私钥对文件（一般来说是文件的哈希值）进行签名。签名值和文件一起放在网站上，因为第三方无法伪造出与我们的私钥，所以他们也无法伪造我们的签名。下载者可以利用与我们私钥对应的公钥对签名进行验证，进而确定文件的真实性。

9.2.4 生成签名

这里简单介绍一下如何生成签名。

签名的长度为 40 字节，分别由两个 20 字节的值组成，分别是 R 和 S ，它们两个的组合 (R, S) 共同组成了 ECDSA 签名。

那又如何来生成这两个值呢？首先，需要生成一个 20 字节的随机值 k ，然后利用椭圆曲线上的“点乘法”获得点 P ，其中 $P=kG$ 。由于 P 点可以由 (x, y) 这样的坐标表示法表示，点 P 的 x 值就是我们需要的 R ，这里 x 和 y 都是 20 字节，得到了 R 之后，接下来我们需要 S 。

为了计算 S 的值，你需要对原始消息进行 SHA1 哈希。哈希的结果是一个 20 字节的值，你可以将它理解为非常大的一个整数，这里记为 z 。得到了 z 之后，我们可以通过以下公式计算 S ：

$$S = k^{-1}(z + dA * R) \bmod p$$

值得注意的是，这里 k^{-1} 的意思是 k 的模逆元。模逆元的性质就是 $(k^{-1} * k) \bmod p$ 等于 1。再来重复一遍这里用到的几个关键信息： k 是我们随机生成的用于计算 R 的一个数， z 是我们消息的哈希值， dA 是我们的私钥， R 是 $k * G$ 的 x 坐标， G 是我们用到的椭圆曲线的指定起点坐标。

9.2.5 验证签名

在得到了签名之后，为了验证它的正确性，你只需要用于签名的私钥所对应的公钥（及椭圆曲线的参数），便可利用下面的公式来计算 P ：

$$P = S^{-1}zG + S^{-1}R * Q_a$$

如果 P 点的 x 坐标的值等于 R 值，则说明签名是有效的，否则签名无效。下面是验证签名的数学证明过程。

因为 $P = S^{-1}zG + S^{-1}R * Q_a$

而 $Q_a = dA * G$ ，所以我们有以下公式：

$$P = S^{-1}zG + S^{-1}R * dAG = S^{-1}(z + dA * R) * G$$

而 P 的 x 坐标应该等于 R ，而 R 是 $k * G$ 的 x 坐标，这意味着：

$$kG = S^{-1}(z + dA * R)G$$

当两边同时移去 G 我们有：

$$k = S^{-1}(z + dA * R)$$

进一步变化得到：

$$S = k^{-1}(z + dA * R)$$

这就是我们之前用于生成 S 的公式，两者相符，说明上面用于验证签名的公式是正确的。

9.3 BCCSP 源码剖析

本节将讲解 BCCSP 区块链加密服务这一模块的设计与实现。

BCCSP 服务涉及的算法：

- ❑ RSA：一种非对称的加密算法，用于加密。有几种族簇，如 RSA1024、RSA2048 等。
- ❑ AES：一种块加密算法，用于加密成块的大量数据。有几种族簇，如 AES128、AES192 等。
- ❑ ECDSA：一种椭圆曲线签名，用于签名。有几种族簇，如 ECDSAP256、ECDSAP384 等。
- ❑ Hash：哈希。有几种族簇，如 SHA256、SHA3_256 等。
- ❑ HMAC：与密匙相关的哈希运算消息认证码。
- ❑ x509：证书的一种，可参看第 12 章中对证书的解释。
- ❑ PKCS#11：一套标准安全接口，可与安全硬件相关。以上算法可以用来进行建立、读取、写入、修改、删除等操作管理。

Fabric 用到的这些技术的常量名称在 `/fabric/bccsp/opts.go` 中开始部分定义，如 ECDSA 支持 ECDSAP256、ECDSAP384 等几种类型。

9.3.1 BCCSP 服务结构

BCCSP 为 Fabric 项目提供各种加密技术、签名技术，MSP 服务模块中就使用到了

BCCSP。BCCSP 服务的代码集中在 /fabric/bccsp 中，目录结构如下：

- ❑ mocks：模拟代码文件夹，可以参看帮助理解 BCCSP 服务。
- ❑ signer：实现的是 crypto 标准库的 Signer 接口，可参看第 12 章中 MSP 服务实现中带“专用签名笔”的身份 signingidentity。该目录的签名接口专用于向外界提供签名对象。
- ❑ factory：BCCSP 服务工厂。
- ❑ pkcs11：BCCSP 服务实现之一——HSM 基础的 BCCSP (the hsm-based BCCSP implementation)。
- ❑ sw：BCCSP 服务实现之二——软件基础的 BCCSP (the software-based implementation of the BCCSP)。
- ❑ utils：BCCSP 服务工具函数。
- ❑ bccsp.go：定义了 BCCSP、Key 接口、众多 BCCSP 接口所使用到的选项接口，如 Key、KeyGenOpts、KeyDerivOpts 等。
- ❑ keystore.go：定义了 key 的管理存储接口，如果生成的 key 不是暂时的，则存储在该接口的实现对象中：如果是暂时性的，则不存储。
- ❑ XXXOpts.go：XXX 表示该目录下的各种值，BCCSP 服务可以使用到的各种技术的选项实现。

从以上可以看出，BCCSP 服务有两种实现：pkcs11 和 sw。其中，pkcs11 是硬件基础的加密服务实现，sw 是软件基础的加密服务实现。这个硬件基础的实现以 <https://github.com/miekg/pkcs11> 这个库为基础，而 HSM 是 Hardware Security Modules 的缩写，即硬件安全模块。相对应的两种 BCCSP 服务实现，这里有两种工厂，两种工厂为其他使用 BCCSP 服务的模块提供了窗口函数（就是给其他模块提供窗口的函数，这些函数一般统一管理自己服务的功能模块，供外界调用，如后文所讲的 InitFactories 函数）。所有产生的 BCCSP 实例存储在 factory/factory.go 所定义的全局变量中。

9.3.2 BCCSP 中的接口和选项

1. 接口

接口相关代码如下：

```
//在fabric/bccsp/bccsp.go中定义
type BCCSP interface {
    //根据key生成选项opts来生成一个key
    //与key有关的选项opts，选项要适合原始的key（与“证书一级一级认证”对应）
    KeyGen(opts KeyGenOpts) (k Key, err error)
    //根据key获取选项opts，从k中重新获取一个key
    KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)
    //根据key导入选项opts，从一个key原始的数据中导入一个key
    KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)
    //根据SKI返回与该接口实例有联系的key
```



```

GetKey(ski []byte) (k Key, err error)
//根据哈希选项opts对一个消息msg进行哈希计算, 如果opts为空, 则使用默认选项
Hash(msg []byte, opts HashOpts) (hash []byte, err error)
//根据哈希选项opts获取hash.Hash实例, 如果opts为空, 则使用默认选项
GetHash(opts HashOpts) (h hash.Hash, err error)
/*根据签名者选项opts, 使用k对digest进行签名, 注意如果需要对一个特别大的消息的hash值*/
//进行签名, 调用者则负责对该特别大的消息进行哈希计算后将其作为digest传入
Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)
//根据鉴定者选项opts, 通过对比k和digest, 鉴定签名
Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)
//根据加密者选项opts, 使用k加密plaintext
Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error)
//根据解密者选项opts, 使用k对ciphertext进行解密
Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error)
}

```

2. 选项

BCCSP 文件夹中任何带“opt”字眼的文件, 都是和选项有关的源码。关于对象的配套选项, 我们在讲 MSP 服务的时候就见识过。根据一个选项的不同配置, 对象主体可以得到不同的数据或进行不同的操作, 这也是一种比较值得学习的语言上的组织技巧, 尤其是在 BCCSP 这种涉及的技术比较多, 而每个对象自身又分为好多类的情况下。在此以哈希选项 HashOpts 和 key 导入选项 KeyImportOpts 作为例子进行说明:

```

//在/fabric/bccsp/bccsp.go中定义
//哈希选项接口
type HashOpts interface {
    Algorithm() string //获取哈希算法字符串标识, 如"SHA256""SHA3_256"
}
//在/fabric/bccsp/hashopts.go中定义
//哈希选项实现之一, SHA256选项
type SHA256Opts struct {
}
func (opts *SHA256Opts) Algorithm() string {
    return SHA256
}
//哈希选项实现之二, SHA384选项
type SHA384Opts struct {
}
func (opts *SHA384Opts) Algorithm() string {
    return SHA384
}
}

-----
//在/fabric/bccsp/bccsp.go中定义
//key导入选项接口
type KeyImportOpts interface {
    Algorithm() string //返回key导入算法字符串标识
    Ephemeral() bool   //如果生成的key是短暂的(ephemeral), 返回true, 否则返回false
}
//在/fabric/bccsp/opts.go中定义

```

```
//key导入选项接口实现之一，ECDSA公匙的导入选项
type ECDSAPKIXPublicKeyImportOpts struct {
    Temporary bool
}
func (opts *ECDSAPKIXPublicKeyImportOpts) Algorithm() string {
    return ECDSA
}
func (opts *ECDSAPKIXPublicKeyImportOpts) Ephemeral() bool {
    return opts.Temporary
}
//key导入选项接口实现之二，ECDSA私匙的导入选项
type ECDSAPrivateKeyImportOpts struct {
    Temporary bool
}
func (opts *ECDSAPrivateKeyImportOpts) Algorithm() string {
    return ECDSA
}
func (opts *ECDSAPrivateKeyImportOpts) Ephemeral() bool {
    return opts.Temporary
}
//比较特殊的，比如签名选项接口SignerOpts
//由于使用的是标准库，因此使用到此选项时多赋值为nil，BCCSP源码中未实现
```

9.3.3 SW 实现方式

BCCSP 的 SoftWare (SW) 实现形式是默认的形式，这点仅从 /fabric/bccsp/factory/opts.go 中工厂的默认选项 DefaultOpts 和核心配置文档中关于 BCCSP 的配置就可以看出来。主要使用的包是标准库 hash 和 crypto (包括其中的各种包，如 aes、rsa、ecdsa、sha256、elliptic、x509 等)。

/fabric/bccsp/sw 目录结构：

- ❑ impl.go: BCCSP 的 SW 实现的 impl。
- ❑ internals.go: 签名者、鉴定者、加密者、解密者接口定义。
- ❑ conf.go: BCCSP 的 SW 实现的配置定义。
- ❑ aes.go: aes 类型的生成 key 函数、加密者 / 解密者实现。
- ❑ ecdsa.go: ecdsa 类型的签名者、公匙 / 私匙鉴定者实现。
- ❑ rsa.go: rsa 类型的签名者、公匙 / 私匙鉴定者实现。
- ❑ aeskey.go: aes 类型的 Key 接口实现。
- ❑ ecdsakey.go: ecdsa 类型的 Key 接口实现。
- ❑ rsakey.go: rsa 类型的 Key 接口实现。
- ❑ dummyks.go: dummy 类型的 KeyStore 接口实现 dummyKeyStore。当生成的 key 是短暂的，则说明这些 key 不会保存到文件中，而是保存到内存中，系统一关闭，这些 key 就消失了。
- ❑ fileks.go: file 类型的 KeyStore 接口实现 fileBasedKeyStore。若生成的 key 不是短

暂的，则说明这些 key 在导入时，会存储在文件中，即便系统关闭，这些 key 也不会消失。

BCCSP 接口实现：

```
//在/fabric/bccsp/sw/impl.go中定义
//SW BCCSP的实例结构体
type impl struct {
    conf *config //BCCSP实例的配置
    ks    bccsp.KeyStore //key存储系统对象，存储和获取Key对象
    encryptors map[reflect.Type]Encryptor //加密者映射
    decryptors map[reflect.Type]Decryptor //解密者映射
    signers    map[reflect.Type]Signer //签名者映射，Key实现的类型作为映射的键
    verifiers  map[reflect.Type]Verifier //鉴定者映射，Key实现的类型作为映射的键
}
//专用生成函数
func New(...) (bccsp.BCCSP, error) { ... }
//接口实现
func (csp *impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, ...) { ... }
...
```

粗线条上看，由于 impl 对象和各种操作选项的存在，绝大部分接口的实现都是以 switch-case 为主干，根据选项的类型或配置，分情况完成功能，且每个分支的操作都类似，如 KeyGen。接下来一一介绍：

- ❑ KeyGen：根据 key 生成的选项不同，生成三种系列的 key，即 ECDSA、AES、RSA。ECDSA 使用库 ecdsa 的 GenerateKey 函数，AES 使用自定义的 GetRandomBytes 函数（在 aes.go 中实现，包装了 rand.Read），RSA 使用库 rsa 的 GenerateKey 函数。每个系列又根据具体参数的不同生成不同“尺寸”的 key，具体的细节略过。最后返回不同的 key 实现对象，如 ecdsaPrivateKey、aesPrivateKey 等（分别定义在同目录下的 XXXkey.go 中）。这里要注意的是，当前版本中用于签名的 key，只支持 ECDSA 系列的 key，这是官方文档中所说的，但是从实现上看签名的支持不止一种。BCCSP 本质上无论实现多少种 key，也只有被调用者能决定使用哪一种。而 MSP 模块是 BCCSP 的使用者之一，应该是在它这个地方只认 ECDSA 的 key。
- ❑ KeyDeriv：根据 key 获取选项 opts，从 k 中重新获取一个 key。这里的重新获取可以理解为把 k 中的内容重新打乱再生成一个 key。处理两种 key 类型，即 ecdsaXXXKey（XXX 代表 Public 和 Private）和 aesPrivateKey。最后返回打乱后重新生成的两种类型的 key，即 reRandomizedKey 和 hmacKey。对于重新生成的 key，如果选项中指定的不是暂时性的 key，则会存储在 ks 中。
- ❑ KeyImport：从原始的数据 raw 中取出选项 opts 指定的 key，如果该 key 不是临时性的，则存储在 ks 中，最后返回 key。这里的原始数据，指的是 []byte 或者含有 key 的数据（如证书数据里的 key）。raw 是一个空接口，也就是说可以接收任何形式的原始数据。为了得到 key，对原始数据 raw 进行转化或抽取，一部分使用了 utils 下的工具函数 utils.Clone 和 utils.DERToPublicKey，如 AES256、ECDSAPrivateKey 等类

型的 key；一部分直接用 Go 语言的断言 `raw.(*Key 类型)`，如 `ECDSAGoPublicKey`、`RSAGoPublicKey` 等类型的 key。

- ❑ Hash：根据哈希选项 `opts` 对一个消息 `msg` 进行哈希计算，返回该 `msg` 的哈希值。如果 `opts` 为空，则使用默认选项，支持 SHA2、SHA3 哈希家族。
- ❑ Sign：根据签名者选项，使用 `k` 对 `digest` 进行签名，这里的签名者选项在当前版本里没什么用处，调用者都给的是 `nil`。签名涉及 `Key` 接口和签名者 `Signer` 在 SW BCCSP 中的实现。`Key` 接口有三种实现，即 `ecdsaKey.go` 中的 `ecdsaPublicKey/ecdsaPrivateKey`、`rsaKey.go` 中的 `rsaPublicKey/rsaPrivateKey` 和 `aesKey.go` 中的 `aesPrivateKey`。这里签名（自然）使用的都是私匙。签名者接口 `Signer` 定义在同目录下的 `internals.go` 中，有两种类型的实现，即 `ecdsa.go` 中的 `ecdsaSigner` 和 `rsa.go` 中的 `rsaSigner`。参看 SW BCCSP 专用生成函数 `New` 的 `signers` 赋值部分，可知用到了两种对应类型的 `Key` 和 `Signer`，即 `ecdsaPrivateKey - ecdsaSigner` 和 `rsaPrivateKey - rsaSigner`。这里签名的实现过程是，首先从该 BCCSP 实例的签名者集成员 `signers` 中获取类型为 `reflect.TypeOf(k)` 的签名者 `signer`；然后直接调用 `signer` 的接口 `Sign`；追溯，`ecdsaSigner` 使用 `ecdsa` 库的 `Sign` 函数，`rsaSigner` 使用 `rsa` 库 `PrivateKey` 结构体的 `Sign` 函数。
- ❑ Verify：根据鉴定者选项 `opts`，通过对比 `k` 和 `digest`，鉴定签名。鉴定涉及 `Key` 接口和鉴定者接口 `Verifier` 在 SW bccsp 中的实现。`Key` 接口实现如上描述。鉴定者接口 `Verifier` 定义在同目录下的 `internals.go` 中，有两种类型的实现，即 `ecdsa.go` 中的 `ecdsaPublicKeyKeyVerifier/ecdsaPrivateKeyVerifier` 和 `rsa.go` 中的 `rsaPublicKeyKeyVerifier/rsaPrivateKeyVerifier`。参看 SW BCCSP 专用生成函数 `New` 的 `verifiers` 赋值部分，可知所有鉴定者（与对应的 `Key` 接口实现）都有用到。这里鉴定的实现过程是，首先从该 BCCSP 实例的鉴定者集成员 `verifiers` 处获取类型为 `reflect.TypeOf(k)` 的鉴定者 `verifier`；然后直接调用 `verifier` 的接口 `Verify`；追溯，`ecdsaXXXKeyVerifier` 使用 `ecdsa` 库的 `Verify` 函数，`rsaXXXKeyVerifier` 使用 `rsa` 库的 `VerifyPSS` 函数（这里 `XXX` 表示 `PublicKey` 或 `Private`）。
- ❑ Encrypt：根据加密者选项 `opts`，使用 `k` 加密 `plaintext`。加密涉及 `*Key` 接口和加密者接口 `Encryptor` 在 SW BCCSP 中的实现。`Key` 接口实现如上描述。加密者接口 `Encryptor` 定义在同目录下的 `internals.go` 中，在 `aes.go` 中实现（只能是 `aes`，因为只有 `aes` 是用来加密的）——`aesCBCPKCS7Encryptor`。参看 SW BCCSP 专用生成函数 `New` 的 `encryptors` 赋值部分，只有 `aesPrivateKey - aesCBCPKCS7Encryptor` 被使用。这里加密的实现过程是，首先从该 BCCSP 实例的加密者集成员 `encryptors` 获取类型为 `reflect.TypeOf(k)` 的加密者 `encryptor`；然后直接调用 `encryptor` 的接口 `Encrypt`；追溯，`aesCBCPKCS7Encryptor` 使用 `aes` 库的加密流程进行加密。
- ❑ Decrypt：解密与加密类似，也是只有 `aes` 配套实现，最终使用 `aes` 库解密流程进行解密。

❑ GetXXX: GetXXX 系列接口，获取实例对象中的数据，具体这里不再详述。

9.3.4 pkcs11 实现方式

BCCSP 的 pkcs11 实现形式主要使用的库与 SW 实现如出一辙，只不过外加一个 github.com/miekg/pkcs11 库，最好参看其文档以熟悉 pkcs11 的简要操作。pkcs11 (Public-Key Cryptography Standards, PKCS) 是一套通用的接口标准，可以说这里是用 pkcs11 实现了 BCCSP 的功能，也为 Fabric 支持热插拔和个人安全硬件模块提供了服务。这点可以从 BCCSP 的 pkcs11 的实现实例的专用生成函数 New (参看下文) 中所调用的 loadLib 函数看出来：loadLib 加载了一个系统中的动态库，能加载系统的动态库，就可以和驱动、热插拔、连接电脑的字符设备联系在一起。比如将来，开发出了一款在区域链上使用的，类似于现在网上银行所用的 U 盾之类的确认个人身份或安全交易的硬件模块或芯片，这些硬件模块或芯片只需要也遵循 pkcs11，Fabric 即可对此进行支持和扩展。

对于 pkcs11 所提供的接口，在此提供两个文档地址，读者可以稍作了解：<https://www.ibm.com/developerworks/cn/security/s-pkcs/> 和 <http://docs.oracle.com/cd/E19253-01/819-7056/6n91eac56/index.html#chapter2-9>。这些文档与 Fabric、区域链无关，但是因为 pkcs11 是通用的接口，所以有一定参考价值。pkcs11 库中的解释相对过于简单。

/fabric/bccsp/pkcs11 目录结构：

- ❑ impl.go: bccsp 的 pkcs11 实现的 impl。
- ❑ conf.go: 定义了 BCCSP 服务的配置和 PKCS11Opts、FileKeystoreOpts、DummyKeystoreOpts 选项。
- ❑ pkcs11.go: 以 pkcs11 库为基础，包装各种 pkcs11 功能，实现了 impl 基于 pkcs11 的内调函数，和一些 BCCSP 服务使用到的独立的内调函数
- ❑ aes.go: 实现 aes 类型的生成 key、加密、解密函数。
- ❑ ecdsa.go: 实现 impl 在 ecdsa 技术下的签名函数 signECDSA 和鉴定函数 verifyECDSA。
- ❑ aeskey.go: 实现 aes 类型的 Key 接口，只实现私匙 aesPrivateKey。
- ❑ ecdsaKey.go: 实现 ecdsa 类型的 Key 接口，实现了公匙 ecdsaPublicKey 和私匙 ecdsaPrivateKey。
- ❑ rsaKey.go: 实现了 rsa 类型的 Key 接口，实现了公匙 rsaPublicKey 和私匙 rsaPrivateKey。
- ❑ dummyks.go: dummy 类型的 KeyStore 接口，实现 DummyKeyStore。
- ❑ fileks.go: file 类型的 KeyStore 接口，实现 FileBasedKeyStore。

BCCSP 接口实现：

```
type impl struct {
    conf *config           //配置
    ks    bccsp.KeyStore      //key存储系统对象，存储和获取Key对象
    ctx   *pkcs11.Ctx           //标准库的pkcs11上下文
    sessions chan pkcs11.SessionHandle //实质是uint，会话标识符频道，默认10个缓存
    slot   uint              //安全硬件外设连接插槽标识号
```



```

lib      string      //加载库所在路径
noPrivImport bool    //禁止导入私匙标识
softVerify bool      //使用软件方式鉴定签名标识
}
//专用生成函数
func New(opts PKCS11Opts, keyStore bccsp.KeyStore) (bccsp.BCCSP, error) { ... }
//接口实现
func (csp *impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, err error) { ... }
...

```

BCCSP 的 pkcs11 实现的骨架在 impl.go 中与 SW 的实现基本一致，只是追溯到最终实现的语句时，SW 实现是使用 crypto 库下的各个包进行签名、加密、密匙导入等；而 pkcs11 则用 pkcs11 包对数据进行了多一层的处理，使用 pkcs11 提供的上下文 (pkcs11.Ctx) 在会话 (SessionHandle) 之上对签名、密匙、加密等进行管理，这也是 pkcs11.go 文件的作用。两者最大的不同是一个面向软件，一个面向硬件。pkcs11 自身又非常冗杂，因此在此只讲 pkcs11 中与安全硬件模块建立连接的 loadLib 函数。

loadLib 函数在 pkcs11.go 中定义，供专用生成函数 New 使用。为建立与安全硬件模块的通信，进行了如下步骤：

(1) 根据所给的系统动态库路径 lib 加载动态库 (如 openCryptoki 的动态库)，调用 pkcs11.New(lib) 建立 pkcs11 实例 ctx。ctx 相当于 Fabric 与安全硬件模块通信的桥梁：bccsp<->ctx<->驱动 lib<->安全硬件模块，只要驱动 lib 是按照 pkcs11 标准开发的。

(2) 对 ctx.Initialize() 进行初始化。从 ctx.GetSlotList(true) 返回的列表中获取由 label 指定的插槽标识 slot (这里的插槽可以简单地理解为电脑主机上供安全硬件模块插入的槽，如 USB 插口，可能不止一个，每一个在系统内核中都有名字和标识号)。

(3) 尝试 10 次调用 ctx.OpenSession 打开一个会话 session (会话就是通过通信路径与安全硬件模块建立连接，可以简单地理解为 pkcs11 的 chan)。

(4) 登录会话 ctx.Login。返回 ctx、slot、会话对象 session，用于赋值给 impl 实例成员 ctx、slot，把 session 发送到 sessions 里。

关于 pkcs11，还有一点可说的，就是 SoftHSM 库，它是一个模拟硬件实现的 pkcs11，对应到系统动态库可参看 impl.go 中 FindPKCS11Lib 测试函数中所涉及的相关内容，如 Linux 下的 libsoftism2.so。现阶段是没有安全硬件模块可以配合测试的，所以只有使用 SoftHSM 模拟测试，将 libsoftism2.so 导入 pkcs11 对象。

BCCSP 工厂对应两种 BCCSP 实现，这里也有两种 bccsp 工厂：pkcs11factory.go 和 swfactory.go。Fabric 中某一模块一旦涉及工厂 factory，则说明该模块基本就是由工厂提供窗口函数，供其他模块调用。这里以 swfactory 为例进行讲解。

/fabric/bccsp/factory/ 目录结构：

❑ factory.go：声明了默认 BCCSP 实例 defaultBCCSP，BCCSP 实例存储映射 bccspMap 等全局变量和这些变量的获取函数 GetXXX。定义 BCCSP 工厂接口。

❑ nopkcs11.go/pkcs11.go：定义了两种版本的工厂选项 FactoryOpts，即初始化工厂函数

InitFactories 和获取指定 BCCSP 实例的函数 GetBCCSPFromOpts。nopkcs11 是默认版本，可根据相应条件编译指定使用哪种版本（编译时加入 nopkcs11 或 !nopkcs11 选项）。两种版本的差异集中在是否使用 pkcs11 上。

- opts.go: 定义了默认的工厂选项 DefaultOpts。
- pkcs11factory.go: pkcs11 类型的 BCCSP 工厂实现 PKCS11Factory。
- swfactory.go: SW 类型的 BCCSP 工厂实现 SWFactory。还定义了 SW 版本的 BCCSP 选项。

swfactory 接口和实现:

```
//在factory.go中定义
//接口
type BCCSPFactory interface {

    //返回工厂的名字
    Name() string
    //返回符合工厂选项opts的BCCSP实例
    Get(opts *FactoryOpts) (bccsp.BCCSP, error)
}

//在swfactory.go中定义
type SWFactory struct{}
func (f *SWFactory) Name() string {
    return SoftwareBasedFactoryName
}
func (f *SWFactory) Get(config *FactoryOpts) (bccsp.BCCSP, error) { ... }
```

实现的代码本身比较简单，Get 最终是调用的 SW 专用生成函数的 New 来生成符合 opts 的 bccsp 实例的。Name 则是直接返回一个 SW 常量。

在每个 chaincode 例子中，如 /fabric/examples/e2e_cli/examples/chaincode/go/chaincode_example02/chaincode_example02.go，都使用了 chaincode 垫片 shim 中的 Start 函数。chaincode 的垫片 shim 核心代码集中在 /fabric/core/chaincode/shim 中，该垫片所承垫的是与各个节点通信的任务，也即 ChaincodeSupport 服务。chaincode 形成的通信信息，通过 shim 分发到各个节点，然后 shim 负责从各个节点收集信息，汇总并返回给 chaincode，完成 chaincode 的功能。其中 shim 的 Start 函数就是用来启动一个 chaincode，其被定义在 /fabric/core/chaincode/shim/chaincode.go 中。在 Start 的函数中，就调用了 err := factory.InitFactories(&factory.DefaultOpts) 来初始化一个默认的 BCCSP 工厂，在此可以知道，这里使用的默认工厂选项（参看 opts.go）就是使用的 swfactory。

Fabric CA 架构设计与讲解

Fabric CA 为 Hyperledger Fabric 提供证书机构的功能。具体来说, Fabric CA 提供以下功能:

- (1) 身份注册, 或者将连接到 LDAP 作为用户注册。
- (2) 颁发登录证书 (ECerts)。
- (3) 证书续期与撤销。

Fabric CA 包含一个服务端组件和一个客户端组件, 稍后会进行介绍。

对贡献 Fabric CA 感兴趣的开发者, 可以参考 Fabric CA repository。

10.1 Fabric CA 用户指南

图 10-1 所示说明了 Fabric CA 服务端如何在 Hyperledger Fabric 架构中发挥作用。

有两种方式可与 Fabric CA 服务端交互: 通过 Fabric CA 客户端或 Fabric SDK。所有与 Fabric CA 的交互都是通过 REST APIs 来实现的。REST APIs 的 swagger 说明文档见 fabric-ca/swagger/swagger-fabric-ca.json。你可以通过 <http://editor2.swagger.io> 在线编辑器来查看这个文档。

Fabric CA 客户端或者 SDK 可能会连接到 Fabric CA 集群中的某一个 Fabric CA 服务端, 这一部分可以参看图 10-1 右上部分。图中客户端连接的是一个 HA 代理节点, 这个 HA 代理节点为 Fabric CA 集群作负载均衡。

所有的 Fabric CA 服务端共享同一个数据库。数据库用来保存用户和证书的信息。如果配置了 LDAP, 那么用户信息将会保存在 LDAP 中, 而不是数据库中。

一个服务端可能包含多个 CA 证书。每一个 CA 证书都是一个根 CA 证书或者一个中间

CA 证书。而每一个中间 CA 证书都有一个根 CA 证书或者其他的一个中间 CA 证书作为其父 CA 证书。

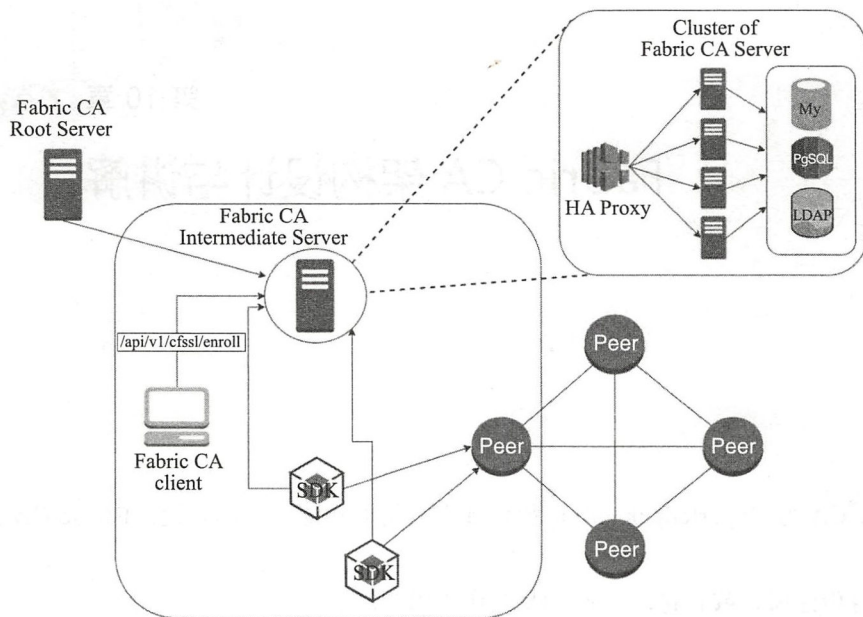


图 10-1 Fabric CA 的作用示意图

入门

下面来体验 Fabric CA。

1. 前置条件

请检查系统是否符合条件：

- ☐ 安装了 Go 1.9+。
- ☐ GOPATH 环境变量设置正确。
- ☐ libtool 和 libtdhl-dev 两个已包安装好。

以下命令用于在 Ubuntu 系统安装 libtool：

```
sudo apt install libtool libltdl-dev
```

以下命令用于在 MacOSX 系统安装 libtool：

```
brew install libtool
```

若要了解更多有关 libtool 的信息，可参考 <https://www.gnu.org/software/libtool/>。

若要了解更多有关 libtdhr-dev 的信息，可参考 https://www.gnu.org/software/libtool/manual/html_node/Using-libltdl.html。

2. 安装

以下命令会在 \$GOPATH/bin 路径下同时安装 fabric-ca-server 和 fabric-ca-client。

```
go get -u github.com/hyperledger/fabric-ca/cmd/...
```

注意：如果你已经下载了 fabric-ca，在运行上述 'go get' 命令时确保当前位于 master 分支，否则你将会看到如下报错：

```
<gopath>/src/github.com/hyperledger/fabric-ca; git pull --ff-only
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
```

```
git pull <remote> <branch>
```

If you wish to set tracking information for this branch you can do so with:

```
git branch --set-upstream-to=<remote>/<branch> tlsdoc
```

```
package github.com/hyperledger/fabric-ca/cmd/fabric-ca-client: exit status 1
```

3. 原生启动服务器

默认配置启动 fabric-ca-server:

```
fabric-ca-server start -b admin:adminpw
```

其中：-b 选项用来提供启动管理员的登录 ID 和密码；如果 LDAP 没有启用 “ldap.enabled” 设置，则 -b 选项是必需的。

默认配置文件 fabric-ca-server-config.yaml 会自动在本地目录创建，这个配置文件可以自定义。

4. 通过 Docker 启动服务器

完成准备工作后，使用命令行来启动服务器。

1) Docker Hub

访问 <https://hub.docker.com/r/hyperledger/fabric-ca/tags/>，找到与你想下载的架构和 fabric-ca 版本相符合的标签。

进入 \$GOPATH/src/github.com/hyperledger/fabric-ca/docker/server 路径并在编辑器中打开 docker-compose.yml。根据之前找到的标签在 image 一行进行修改。beta 版本 x86 架构的 docker-compose.yml 文件如下所示。

```
fabric-ca-server:
  image: hyperledger/fabric-ca:x86_64-1.0.0-beta
  container_name: fabric-ca-server
  ports:
    - "7054:7054"
  environment:
    - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
```

```
volumes:
  - "../fabric-ca-server:/etc/hyperledger/fabric-ca-server"
command: sh -c 'fabric-ca-server start -b admin:adminpw'
```

在 docker-compose.yml 文件的根目录下打开一个终端并且运行如下命令：

```
# docker-compose up -d
```

如果指定的 fabric-ca 镜像之前并不存在，这个命令将对其进行下载。这个命令还会启动 fabric-ca 服务端的一个实例。

2) 构建你自己的 Docker 镜像

你可以通过 docker-compose 构建和启动服务器，具体如下所示：

```
cd $GOPATH/src/github.com/hyperledger/fabric-ca
make docker
cd docker/server
docker-compose up -d
```

hyperledger/fabric-ca docker 镜像包含 fabric-ca-server 和 fabric-ca-client。

5. 体验 Fabric CA 命令行

这一部分提供 fabric-ca-server 和 fabric-ca-client 命令行的使用说明。其他使用信息会在接下来的内容中提供。

fabric-ca-server:

Hyperledger Fabric Certificate Authority Server

Usage:

```
fabric-ca-server [command]
```

Available Commands:

```
init      Initialize the fabric-ca server
start     Start the fabric-ca server
version   Prints Fabric CA Server version
```

Flags:

```
--address string          Listening address of
                           fabric-ca-server (default "0.0.0.0")
-b, --boot string         The user:pass for
                           bootstrap admin which is required to build default config file
--ca.certfile string      PEM-encoded CA
                           certificate file (default "ca-cert.pem")
--ca.chainfile string     PEM-encoded CA chain file
                           (default "ca-chain.pem")
--ca.keyfile string       PEM-encoded CA key file
-n, --ca.name string      Certificate Authority
                           name
--cacount int             Number of non-default CA
                           instances
--cafiles stringSlice     A list of comma-separated CA
```

```

configuration files
--cfg.affiliations.allowremove          Enables removal of
affiliations dynamically
--cfg.identities.allowremove            Enables removal of identities
dynamically
--crl.expiry duration                    Expiration for the CRL
generated by the gencrl request (default 24h0m0s)
--crlsizelimit int                       Size limit of an acceptable
CRL in bytes (default 512000)
--csr.cn string                          The common name field of the
certificate signing request to a parent fabric-ca-server
--csr.hosts stringSlice                  A list of space-separated host
names in a certificate signing request to a parent fabric-ca-server
--csr.serialnumber string                The serial number in a
certificate signing request to a parent fabric-ca-server
--db.datasources string                  Data source which is database
specific (default "fabric-ca-server.db")
--db.tls.certfiles stringSlice           A list of comma-separated PEM-
encoded trusted certificate files (e.g. root1.pem,root2.pem)
--db.tls.client.certfile string          PEM-encoded certificate file
when mutual authenticate is enabled
--db.tls.client.keyfile string           PEM-encoded key file when
mutual authentication is enabled
--db.type string                         Type of database; one of:
sqlite3, postgres, mysql (default "sqlite3")
-d, --debug                             Enable debug level logging
-H, --home string                        Server's home directory
(default "/etc/hyperledger/fabric-ca")
--intermediate.enrollment.label string   Label to use in HSM
operations
--intermediate.enrollment.profile string  Name of the signing
profile to use in issuing the certificate
--intermediate.parentserver.caname string Name of the CA to connect
to on fabric-ca-server
-u, --intermediate.parentserver.url string URL of the parent fabric-
ca-server (e.g. http://<username>:<password>@<address>:<port>)
--intermediate.tls.certfiles stringSlice A list of comma-separated
PEM-encoded trusted certificate files (e.g. root1.pem,root2.pem)
--intermediate.tls.client.certfile string PEM-encoded certificate
file when mutual authenticate is enabled
--intermediate.tls.client.keyfile string PEM-encoded key file when
mutual authentication is enabled
--ldap.attribute.names stringSlice       The names of LDAP
attributes to request on an LDAP search
--ldap.enabled                           Enable the LDAP client
for authentication and attributes
--ldap.groupfilter string                 The LDAP group filter for
a single affiliation group (default "(memberUid=%s)")
--ldap.tls.certfiles stringSlice          A list of comma-separated
PEM-encoded trusted certificate files (e.g. root1.pem,root2.pem)
--ldap.tls.client.certfile string         PEM-encoded certificate

```



```

        file when mutual authenticate is enabled
--ldap.tls.client.keyfile string          PEM-encoded key file when
        mutual authentication is enabled
--ldap.url string                        LDAP client URL of form
        ldap://adminDN:adminPassword@host[:port]/base
--ldap.userfilter string                The LDAP user filter to
        use when searching for users (default "(uid=%s)")
-p, --port int                          Listening port of fabric-
ca-server (default 7054)
--registry.maxenrollments int           Maximum number of
        enrollments; valid if LDAP not enabled (default -1)
--tls.certfile string                  PEM-encoded TLS
        certificate file for server's listening port (default "tls-cert.
        pem")
--tls.clientauth.certfiles stringSlice  A list of comma-separated
        PEM-encoded trusted certificate files (e.g. root1.pem,root2.pem)
--tls.clientauth.type string           Policy the server will
        follow for TLS Client Authentication. (default "noclientcert")
--tls.enabled                          Enable TLS on the
        listening port
--tls.keyfile string                  PEM-encoded TLS key for
        server's listening port

```

Use "fabric-ca-server [command] --help" for more information about a command.

fabric-ca-client:

Hyperledger Fabric Certificate Authority Client

Usage:

```
fabric-ca-client [command]
```

Available Commands:

```

affiliation Manage affiliations
enroll      Enroll an identity
gencrl      Generate a CRL
gencsr      Generate a CSR
getcacert   Get CA certificate chain
identity    Manage identities
reenroll    Reenroll an identity
register     Register an identity
revoke      Revoke an identity
version     Prints Fabric CA Client version

```

Flags:

```

--caname string      Name of CA
--csr.cn string      The common name field of the certificate
                    signing request
--csr.hosts stringSlice  A list of space-separated host names in
                    a certificate signing request
--csr.names stringSlice  A list of comma-separated CSR names of

```

```

    the form <name>=<value> (e.g. C=CA,O=Org1)
--csr.serialnumber string      The serial number in a certificate
    signing request
-d, --debug                  Enable debug level logging
--enrollment.attrs stringSlice A list of comma-separated attribute
    requests of the form <name>[:opt] (e.g. foo,bar:opt)
--enrollment.label string     Label to use in HSM operations
--enrollment.profile string    Name of the signing profile to use in
issuing the certificate
-H, --home string             Client's home directory (default
"$HOME/.fabric-ca-client")
--id.affiliation string        The identity's affiliation
--id.attrs stringSlice         A list of comma-separated attributes of
    the form <name>=<value> (e.g. foo=foo1,bar=bar1)
--id.maxenrollments int        The maximum number of times the secret
    can be reused to enroll (default CA's Max Enrollment)
--id.name string               Unique name of the identity
--id.secret string             The enrollment secret for the identity
    being registered
--id.type string               Type of identity being registered (e.g.
    'peer, app, user') (default "client")
-M, --mspsdir string           Membership Service Provider directory
    (default "msp")
-m, --myhost string            Hostname to include in the certificate
    signing request during enrollment (default "$HOSTNAME")
-a, --revoke.aki string        AKI (Authority Key Identifier) of the
    certificate to be revoked
-e, --revoke.name string        Identity whose certificates should be
    revoked
-r, --revoke.reason string      Reason for revocation
-s, --revoke.serial string      Serial number of the certificate to be
    revoked
--tls.certfiles stringSlice    A list of comma-separated PEM-encoded
    trusted certificate files (e.g. root1.pem,root2.pem)
--tls.client.certfile string    PEM-encoded certificate file when mutual
    authenticate is enabled
--tls.client.keyfile string      PEM-encoded key file when mutual
    authentication is enabled
-u, --url string                URL of fabric-ca-server (default
"http://localhost:7054")

```

Use "fabric-ca-client [command] --help" for more information about a command.



注意 在命令行中需要给某个选项输入列表时，可以用空格分割，或者多次使用该选项。例如，指定 host1 和 host2 给 csr.hosts 选项，你可以用 -csr.hosts 'hosts1,hosts2' 或者 -csr.hosts host1 -csr.hosts host2。当你使用前一种方式时，注意命令中逗号前后没有空格。

6. 配置设置

Fabric CA 提供三种方法来配置 Fabric CA 服务端和客户端，三种方法的优先级如下：

- (1) 命令行参数；
- (2) 环境变量；
- (3) 配置文件。

接下来，我们试着对配置文件进行修改。配置文件的修改可以被环境变量或者命令行参数覆盖。例如，假设我们有以下客户端的配置文件：

```
tls:
  # Enable TLS (default: false)
  enabled: false

  # TLS for the client's listening port (default: false)
  certfiles:
  client:
    certfile: cert.pem
    keyfile:
```

下述环境变量可以被用于覆盖上述配置文件中的 `cert.pem` 设置。

```
export FABRIC_CA_CLIENT_TLS_CLIENT_CERTFILE=cert2.pem
```

如果我们想把环境变量和配置文件都覆盖，可以使用如下命令行：

```
fabric-ca-client enroll --tls.client.certfile cert3.pem
```

上述同样适用于 `fabric-ca-server`，唯一的区别是用于 `fabric-ca-server` 时将采用 `FABRIC_CA_SERVER` 替换 `FABRIC_CA_SERVER` 作为环境变量的前缀。

7. 文件路径说明

Fabric CA 服务端和客户端配置文件中所有属性都支持相对路径和绝对路径。相对路径是指相对于配置目录。例如，如果配置目录是 `~/config`，并且 `tls` 部分如下所示，Fabric CA 服务端或客户端将在 `~/config` 目录下寻找 `root.pem` 文件，在 `~/config/certs` 目录下寻找 `cert.pem` 文件，在 `/abs/path` 目录下寻找 `key.pem` 文件。

```
tls:
  enabled: true
  certfiles:
    - root.pem
  client:
    certfile: certs/cert.pem
    keyfile: /abs/path/key.pem
```

10.2 Fabric-CA-Server

这一节介绍 Fabric CA 服务端。

在启动 Fabric CA 服务端之前，你可以先初始化 Fabric CA 服务端。通过初始化，程序会自动生成一份默认配置文件，方便用户在启动前自行修改一些配置项。

Fabric CA 服务端的根目录定义规则如下：

- ❑ 如果 `-home` 命令行参数已经设置，则使用已设置的参数。
- ❑ 否则，如果存在环境变量 `FABRIC_CA_SERVER_HOME`，则使用该环境变量。
- ❑ 否则，如果存在环境变量 `FABRIC_CA_HOME`，则使用该环境变量。
- ❑ 否则，如果存在环境变量 `CA_CFG_PATH`，则使用该环境变量。
- ❑ 否则，使用当前的工作目录作为根目录。

本章接下来的部分，我们假设你已经设置了环境变量 `FABRIC_CA_HOME`，其值设置为 `$HOME/fabric-ca/server`。

接下来的内容都默认服务端配置文件存在于服务端根目录下。

10.2.1 初始化服务端

用以下命令初始化 Fabric CA 服务端：

`fabric-ca-server init -b admin:adminpw`：当 LDAP 无效时，初始化必须提供 `-b` (bootstrap identity：引导身份) 选项。启动服务端时，至少需要提供一个引导身份。这个身份将是服务端管理员。服务端配置文件包含一个可以配置的证书签名请求 (Certificate Signing Request, CSR)。下面是一个 CSR 的例子。

- ❑ `cn: fabric-ca-server`
- ❑ `names:`
 - `C: US // 国家`
 - `ST: "North Carolina" // state`
 - `L: // 城市或 location`
 - `O: Hyperledger // 组织名字`
 - `OU: Fabric // 组织单位`
- ❑ `hosts:`
 - `host1.example.com`
 - `localhost`
- ❑ `ca:`
 - `expiry: 131400h`
 - `pathlength: 1`

上面所有的字段都符合 X.509 签名与证书规范，可以由 `fabric-ca-server init` 命令来生成。服务端配置文件中提到的 `ca.certfile` 和 `ca.keyfile` 这两个文件也符合 X.509 规范。字段如下：

- ❑ `cn` 通用名；

- ❑ 组织;
- ❑ OU 组织单位;
- ❑ L 地址或城市;
- ❑ ST 州 (省);
- ❑ C 国家。

如果需要修改 CSR 里面的值, 你可以修改配置文件, 然后把配置中由 `ca.certfile` 和 `ca.keyfile` 这两项指明的文件删除。然后再运行一次 `fabric-ca-server init -b admin:adminpw` 命令。

`fabric-ca-server init` 命令生成一个自签名的 CA 证书, 除非使用了 `-u <parent-fabric-ca-server-URL>` 选项。如果使用了 `-u` 选项, 本服务端 CA 证书会由父 Fabric CA 服务端签名。为了向父 Fabric CA 服务端验证身份, URL 格式必须为 `<scheme>://<enrollmentID>:<secret>@<host>:<port>`, 其中 `<enrollmentID>` 和 `<secret>` 对应着一个 `'hf.IntermediateCA'` 属性为 `true` 的身份。`fabric-ca-server init` 命令还会在服务器根目录生成一个名为 `fabric-ca-server-config.yaml` 的默认配置文件。

如果想要 Fabric CA 服务端使用你提供的 CA 签名证书和密钥文件, 你必须把文件分别放在由 `ca.certfile` 和 `ca.keyfile` 引用的位置。两个文件都必须用 PEM 编码并且不能加密。具体来说, CA 证书文件的内容必须以 “----- BEGIN CERTIFICATE -----” 开头, 并且密钥文件的内容必须以 “----- BEGIN PRIVATE KEY -----” 开头, 而不是 “-----BEGIN ENCRYPTED PRIVATE KEY-----”。

10.2.2 算法和密钥长度

可以通过自定义 CSR 来生成支持 ECDSA 和 RSA 的 X.509 证书和密钥。以下是一个椭圆曲线数字签名算法 (ECDSA) 的设置, 采用的曲线是 `prime256v1`, 签名算法是 `ecdsa-with-SHA256`。

```
key:
  algo: ecdsa
  size: 256
```

对算法和密钥长度的选择取决于你对安全的考量。
ECDSA 提供以下选项如表 10-1 所示。

表 10-1 ECDSA 选项表

size	ASN1 OID	Signature Algorithm
256	prime256v1	ecdsa-with-SHA256
384	secp384r1	ecdsa-with-SHA384
521	secp521r1	ecdsa-with-SHA512

10.2.3 启动服务端

启动 Fabric CA 服务端：

```
fabric-ca-server start -b <admin>:<adminpw>
```

如果服务器之前没有初始化过，它会在第一次启动的时候进行初始化。在初始化的过程中，如果 `ca-cert.pem` 和 `ca-key.pem` 这两个文件不存在，它会生成这两个文件；如果配置文件不存在，它会生成默认配置文件。

除非 Fabric CA 服务端配置了使用 LDAP，否则它必须配置至少一个预注册引导身份来允许你登录其他身份。`-b` 选项指明引导身份的用户名和密码。

为了使 Fabric Ca Server 监听 HTTPS 而不是 HTTP，配置 `tls.enabled` 为 `true`。

为了限制登录时相同密码的次数，可以在配置文件中设置 `registry.maxEnrollments` 为恰当的值。如果你设置这个值为 1，Fabric CA 服务端只允许一个密码被一个登录 ID 使用（即不会出现多个 ID 有相同密码的情况）。如果你设置这个值为 -1（默认值就为 -1），Fabric CA 服务端不会限制密码的重复使用次数；如果你设置这个值为 0，Fabric CA 服务端将会禁止所有身份的登录和注册。

Fabric CA 服务端现在应该正在监听 7054 端口。

10.2.4 配置数据库

这一节讲解如何配置 Fabric CA 服务端并连接到 Postgres 或者 MySQL。默认的数据库是 SQLite，默认的数据库文件是 `fabric-ca-server.db`，数据库文件存放在 Fabric CA 服务端的根目录。

你想运行 Fabric CA 服务端集群，可以照下面的指引配置 Postgres 或者 MySQL。Fabric CA 在群集设置中支持以下数据库版本：

- ❑ PostgreSQL9.5.5 或者更高版本；
- ❑ MySQL5.7 或者更高版本。

10.2.5 PostgreSQL

下面的例子可以添加到服务端的配置文件中，来使服务端连接到一个 Postgres 数据库。别忘了正确地自定义各种参数。

```
db:
  type: postgres
  datasource: host=localhost port=5432 user=Username password=Password dbname=
             fabric-ca-server sslmode=verify-full
```

指定 `sslmode` 来配置 SSL 认证的类型。`sslmode` 有效的值如表 10-2 所示。

表 10-2 Sslmode 有效值表

Mode	Description
disable	No SSL
require	Always SSL (skip verification)
verify-ca	Always SSL (verify that the certificate presented by the server was signed by a trusted CA)
verify-full	Same as verify-ca AND verify that the certificate presented by the server was signed by a trusted CA and the server hostname matches the one in the certificate

如果你想使用 TLS，那么需要在配置文件中指明 db.tls。如果 Postgres 服务器开启了 SSL 客户端认证，那么客户端的证书和密钥文件必须在 db.tls.client 中指明。下面是 db.tls 部分的一个例子：

```
db:
  ...
  tls:
    enabled: true
    certfiles:
      - db-server-cert.pem
    client:
      certfile: db-client-cert.pem
      keyfile: db-client-key.pem
```

其中：certfiles 为可信任的根证书文件列表，采用 PEM 编码；certfile 和 keyfile 为用于与 Postgres 服务器安全通信的证书和密钥文件，采用 PEM 编码。


10.2.6 PostgreSQL SSL 配置

在 PostgreSQL 服务器上配置 SSL 的基本说明：

(1) 在 postgresql.conf 中，取消注释 SSL 并设置为 “on” (SSL = on)。

(2) 将证书和密钥文件放在 PostgreSQL data 目录中。

有关生成自签名证书的说明可参见 <https://www.postgresql.org/docs/9.5/static/ssl-tcp.html>。

 **注意** 自签名证书仅用于测试目的，不应在生产环境中使用。

PostgreSQL 服务器需要客户端证书。

(1) 将你信任的证书颁发机构 (CA) 的证书放入 PostgreSQL data 目录中的文件 root.crt 中。

(2) 在 postgresql.conf 中，将 “ssl_ca_file” 设置为指向客户端的根证书 (CA cert)。

(3) 在 pg_hba.conf 中的相应 hostssl 行上将 clientcert 参数设置为 1。

有关在 PostgreSQL 服务器上配置 SSL 的更多详细信息，请参阅 <https://www.postgresql.org/docs/9.5/static/ssl-tcp.html>。

org/docs/9.4/static/libpq-ssl.html。

10.2.7 MySQL

下面的例子可以添加到 Fabric CA 服务端配置文件，用来连接到 MySQL 数据库。别忘了正确地自定义各种参数。

在 MySQL 5.7.X 中，某些模式会影响服务器是否允许“0000-00-00”作为有效日期。我们有必要放宽 MySQL 服务器使用的模式。我们希望服务器能够接受零日期值。

在 my.cnf 中，找到配置选项 sql_mode 并删除 NO_ZERO_DATE（如果存在）。进行此更改后重新启动 MySQL 服务器。

请参阅以下有关不同可用模式的 MySQL 文档，并为正在使用的特定版本的 MySQL 选择适当的设置：<https://dev.mysql.com/doc/refman/5.7/en/sql-mode.html>

db:

```
type: mysql
datasource: root:rootpw@tcp(localhost:3306)/fabric_ca?parseTime=true&tls=custom
```

如果要使用 TLS，需要配置 db.tls.client 部分，具体参考 10.2.5 节。

10.2.8 MySQL SSL 配置

在 MySQL 服务器上配置 SSL：

(1) 打开或创建 my.conf 文件，在 [mysqld] 部分添加或取消注释下面的行。这些应指向服务器的密钥和证书以及根 CA 证书。有关创建服务器和客户端认证的说明可参阅 <http://dev.mysql.com/doc/refman/5.7/en/creating-ssl-files-using-openssl.html>。

```
[mysqld] ssl-ca=ca-cert.pem ssl-cert=server-cert.pem ssl-key=server-key.pem
```

(2) 可以运行以下查询确认 SSL 已启用。

```
mysql> SHOW GLOBAL VARIABLES LIKE 'have_%ssl';
```

此时应该看到表 10-3 所示的各项。

表 10-3 SSL 确认表

Variable_name	Value	Variable_name	Value
have_openssl	YES	have_ssl	YES

(3) 服务器端 SSL 配置完成后，下一步是创建一个有权通过 SSL 访问的 MySQL 服务器用户。为此，请登录到 MySQL 服务器，然后输入：

```
mysql> GRANT ALL PRIVILEGES ON . TO 'ssluser'@'%' IDENTIFIED BY 'password'
        REQUIRE SSL; mysql> FLUSH PRIVILEGES;
```

(4) 如果想给出用户访问服务器的特定 IP 地址，应将 '%' 更改为特定的 IP 地址。

MySQL 服务器需要客户端证书

安全连接的选项与服务器端使用的选项类似。

❑ `ssl-ca` : 验证证书颁发机构颁发 (CA) 证书。该选项 (如果使用) 必须指定与服务器使用的相同证书。

❑ `ssl-cert`: 验证 MySQL 服务器的证书。

❑ `ssl-key`: 验证 MySQL 服务器的私钥。

假设你要使用无特殊加密要求的账户进行连接, 或者使用包含 `REQUIRE SSL` 选项的 `GRANT` 语句创建该账户。作为推荐的一组安全连接选项, 至少使用 `-ssl-cert` 和 `-ssl-key` 选项来启动 MySQL 服务器。然后在服务器配置文件中设置 `db.tls.certfiles` 属性并启动 Fabric CA 服务器。若要指定客户端证书, 使用 `require X509` 选项创建账户。然后客户端还必须指定适当的客户端密钥和证书文件, 否则 MySQL 服务器将拒绝连接。要为 Fabric CA 服务端指定客户端密钥和证书文件, 可设置 `db.tls.client.cert` 和 `db.tls.client.keyfile` 的属性。

10.2.9 配置 LDAP

Fabric CA 服务端可以配置为连接到一个 LDAP 服务器。Fabric CA 服务端可以连接到一个 LDAP 服务器来做下面的事情:

❑ 登录前验证一个身份。

❑ 授权时获取一个身份的属性值。

在配置文件中修改 LDAP 的配置来连接到一个 LDAP 服务器。

ldap:

```
# Enables or disables the LDAP client (default: false)
enabled: false
# The URL of the LDAP server
url: <scheme>://<adminDN>:<adminPassword>@<host>:<port>/<base>
userfilter: <filter>
attribute:
  # 'names' is an array of strings that identify the specific attributes
  # which are requested from the LDAP server.
  names: <LDAPAttrs>
  # The 'converters' section is used to convert LDAP attribute values
  # to fabric CA attribute values.
  #
  # For example, the following converts an LDAP 'uid' attribute
  # whose value begins with 'revoker' to a fabric CA attribute
  # named "hf.Revoker" with a value of "true" (because the expression
  # evaluates to true).
  #   converters:
  #     - name: hf.Revoker
  #       value: attr("uid") =~ "revoker*"
  #
  # As another example, assume a user has an LDAP attribute named
```



```

# 'member' which has multiple values of "dn1", "dn2", and "dn3".
# Further assume the following configuration.
#   converters:
#     - name: myAttr
#       value: map(attr("member"), "groups")
#   maps:
#     groups:
#       - name: dn1
#         value: orderer
#       - name: dn2
#         value: peer
# The value of the user's 'myAttr' attribute is then computed to be
# "orderer,peer,dn3". This is because the value of 'attr("member")' is
# "dn1,dn2,dn3", and the call to 'map' with a 2nd argument of
# "group" replaces "dn1" with "orderer" and "dn2" with "peer".
converters:
  - name: <fcaAttrName>
    value: <fcaExpr>
maps:
  <mapName>:
    - name: <from>
      value: <to>

```

其中：

- ❑ scheme: 可为 ldap 或者 ldaps。
- ❑ adminDN: 管理员的区别名。
- ❑ pass: 管理员的密码。
- ❑ host: LDAP 服务器的域名或者 IP。
- ❑ port: 可选的端口号, ldap 默认为 389, ldaps 默认为 636。
- ❑ base: 可选的 LDAP 树的根, 用于搜索时。
- ❑ filter: 搜索时的过滤器, 把登录用户名转换为一个区别名。比如, (uid=%s) 会搜索 uid 值等于用户登录名的 LDAP 实体。与此类似, (email=%s) 可以用于邮箱地址作为用户名的登录。
- ❑ LDAPAttrs: 代表用户从 LDAP 服务器请求的 LDAP 属性名称数组。
- ❑ converters: 用于将 LDAP 属性转换为 Fabric CA 属性, 其中 fcaAttrName 是 Fabric CA 属性的名称; fcaExpr 是一个表达式, 它的值将被分配给 fabric CA 属性。例如, 假设是 ["uid"], 是 'hf.Revoker', 是 'attr("uid") =~ "revoker*"'。那么这意味着用户从 LDAP 服务器请求名为 "uid" 的属性。如果用户 'uid' LDAP 属性的值以 'revoker' 开头, 那么用户将赋予 'hf.Revoker' 属性的值为 'true'; 否则, 用户将赋予 'hf.Revoker' 属性的值为 'false'。
- ❑ maps 用于映射 LDAP 响应值。典型的用例是将与 LDAP 组关联的独特名称映射到身份类型。

LDAP 表达式语言使用 <https://github.com/Knetic/govaluate/blob/master/MANUAL.md> 中描述的 govaluate 包。这定义了诸如 “~” 之类的运算符和诸如 “revoker *” 之类的文字，这是一个正则表达式。扩展基础 govaluate 语言的特定 LDAP 变量和函数如下：

- ❑ DN：等于用户名称的变量。
- ❑ affiliation：等于用户隶属关系的变量。
- ❑ attr：一个需要 1 或 2 个参数的函数。第一个参数是 LDAP 属性名称。第二个参数是一个分隔符字符串，用于将多个值连接到单个字符串中，默认分隔符字符串是 “,”。attr 函数总是返回一个 'string' 类型的值。
- ❑ map：一个需要 2 个参数的函数。第一个参数是任何字符串。第二个参数是一个映射的名字，用来对第一个参数的字符串进行替换。
- ❑ if：一个需要 3 个参数的函数。其中第一个参数必须解析为布尔值。如果它的计算结果为 true，则返回第二个参数；否则，返回第三个参数。

例如，如果用户具有以 “O=org1, C=US” 结尾的名称，或者用户具有以 “org1.dept2” 开头的隶属关系，且 “admin” 属性的值为 “true”，那么下述表达式计算结果将为 true。

```
DN =~ "O=org1,C=US" || (affiliation =~ "org1.dept2." && attr('admin') = 'true')
```

由于 attr 函数总是返回 'string' 类型的值，因此数值运算符将不会用于构造表达式。例如，以下不是有效的表达式：

```
value: attr("gidNumber") >= 10000 && attr("gidNumber") < 10006
```

另外，下述表达式中引号里的正则表达式可以被用于返回一个与上述表达式相等的结果。

```
value: attr("gidNumber") =~ "1000[0-5]$" || attr("mail") == "root@example.com"
```

以下是 OpenLDAP 采用默认设置的例子，OpenLDAP 的 docker 镜像在 <https://github.com/osixia/docker-openldap> 中。

```
ldap:
  enabled: true
  url: ldap://cn=admin,dc=example,dc=org:admin@localhost:10389/dc=example,dc=org
  userfilter: (uid=%s)
```

在 FABRIC_CA/scripts/run-ldap-tests 中有一个脚本，这个脚本能启动 OpenLDAP 的 docker 镜像，配置 LDAP，然后运行 FABRIC_CA/cli/server/ldap/ldap_test.go 里面的 LDAP 测试，以验证 LDAP 配置是否成功，最后停止 OpenLDAP 服务器。

当 LDAP 配置好后，登录的流程如下：

- ❑ 向 Fabric CA 客户端或者客户端 SDK 发送一个登录请求，带上 basic 方式的授权头。
- ❑ Fabric CA 服务端收到登录请求，解码授权头里的身份名和密码，查找与身份名相关联的区别名（关联方式在配置文件里的 “userfilter” 中定义），然后用身份密码尝试绑定一个 LDAP。如果 LDAP 绑定成功，那么说明登录过程被批准了，能够继续。

10.2.10 构建一个集群

你可以使用任意 IP 代理来为 Fabric CA 服务端集群做负载均衡。这一节提供了一个例子来介绍如何使用 Haproxy 来为集群路由。别忘了修改域名和端口。

haproxy.conf:

```
global
    maxconn 4096
    daemon

defaults
    mode http
    maxconn 2000
    timeout connect 5000
    timeout client 50000
    timeout server 50000

listen http-in
    bind *:7054
    balance roundrobin
    server server1 hostname1:port
    server server2 hostname2:port
    server server3 hostname3:port
```



注意 如果使用 TLS，需要使用 mode tcp。

10.2.11 构建多个 CA

Fabric CA 服务端默认由一个默认的 CA 组成。但是，可以使用 `cafiles` 或 `cacount` 配置选项将其他的 CA 添加到单个服务端。每个额外的 CA 将拥有自己的主目录。

1. cacount

`cacount` 提供了一种快速启动多个默认额外 CA 的方法，此时主目录将相对于服务器目录。使用此选项，目录结构如下所示：

```
--<Server Home>
|--ca
|   |--ca1
|   |--ca2
```

每个额外的 CA 将在其主目录中获得一个默认配置文件，在配置文件中它将包含一个唯一的 CA 名称。

例如，以下命令将启动 2 个默认 CA 实例：

```
fabric-ca-server start -b admin:adminpw --cacount 2
```


2. cafiles

如果在使用 `cafiles` 配置选项时未提供绝对路径，则 CA 主目录将相对于服务器目录。

要使用此选项，必须已经为每一个要启动的 CA 生成并配置 CA 配置文件。每一个配置文件必须具有唯一的 CA 名称和通用名称（CN），否则服务器将无法启动，因为这些名称必须是唯一的。CA 配置文件将覆盖任何默认 CA 配置，并且 CA 配置文件中的任何缺失选项都会被默认 CA 的值替换。

CA 配置优先级如下：

- (1) CA 配置文件；
- (2) 默认 CA 命令行参数；
- (3) 默认 CA 环境变量；
- (4) 默认 CA 配置文件。

一个 CA 配置文件至少应包括以下内容：

```
ca:
# Name of this CA
name: <CANAME>

csr:
  cn: <COMMONNAME>
```

你可以将你的目录结构按照下述方法进行配置：

```
--<Server Home>
|--ca
|  |--ca1
|    |-- fabric-ca-config.yaml
|  |--ca2
|    |-- fabric-ca-config.yaml
```

如下命令将启动两个自定义的 CA 实例：

```
fabric-ca-server start -b admin:adminpw --cafiles ca/ca1/fabric-ca-config.yaml
--cafiles ca/ca2/fabric-ca-config.yaml
```

10.2.12 登录一个中间 CA

为了给中间 CA 创建一个 CA 签名证书，中间 CA 必须和一个父 CA 一起登录，就像一个 `fabric-ca-client` 必须和一个 CA 一起登录。如下命令所示，这些可以通过 `-u` 选项指定父 CA 的 URL、登录 ID 以及登录密码来完成。跟这个登录 ID 相关联的身份必须拥有名为“`hf.IntermediateCA`”且值为“`true`”的属性。登录 ID 设置已颁发证书的 CN。如果中间 CA 尝试明确指定 CN 的值，则会发生错误。

```
fabric-ca-server start -b admin:adminpw -u http://<enrollmentID>:<secret>@<paren
tserver>:<parentport>
```

10.2.13 升级服务端

升级 Fabric CA 客户端之前，必须升级 Fabric CA 服务端。升级之前建议备份当前数据库：

- ❑ 如果使用 sqlite3，请备份当前数据库文件（默认情况下命名为 fabric-ca-server.db）。
- ❑ 对于其他数据库类型，请使用适当的备份 / 复制机制。

升级 Fabric CA 服务端的单个实例：

- (1) 停止 fabric-ca-server 进程。
- (2) 确保当前数据库已备份。
- (3) 将之前的 fabric-ca-server 二进制文件替换为升级版本。
- (4) 启动 fabric-ca-server 进程。
- (5) 使用以下命令验证 fabric-ca-server 进程是否可用，其中 <host> 用于启动服务器的主机名：

```
fabric-ca-client getcacert -u http://<host>:7054
```

10.2.14 升级一个集群

要使用 MySQL 或 Postgres 数据库升级 fabric-ca-server 实例的集群，请执行以下步骤。我们假设你正在使用 haproxy 为 host1 和 host2 上 fabric-ca-server 集群（分别监听 7054 端口）做负载均衡。在完成升级后，你将为 host3 和 host4 上升级后的 fabric-ca-server 集群（分别监听 7054 端口）作负载均衡。

为了使用 haproxy stats 监视更改、启用统计信息收集，将以下行添加到 haproxy 配置文件的全局部分：

```
stats socket /var/run/haproxy.sock mode 666 level operator
stats timeout 2m
```

重启 haproxy 来应用更改：

```
# haproxy -f <configfile> -st $(pgrep haproxy)
```

要显示 haproxy “show stat” 命令的摘要信息，以下功能可能对解析返回的大量 CSV 数据有用：

```
haProxyShowStats() {
    echo "show stat" | nc -U /var/run/haproxy.sock | sed '1s/^# *///' |
    awk -F',' -v fmt="%4s %12s %10s %6s %6s %4s %4s\n" '
    { if (NR==1) for (i=1;i<=NF;i++) f[tolower($i)]=i }
    { printf fmt, $f["sid"],$f["pxname"],$f["svname"],$f["status"],
      $f["weight"],$f["act"],$f["bck"] } }'
}
```

最初，你的 haproxy 配置文件与以下内容类似：

```
server server1 host1:7054 check
server server2 host2:7054 check
```

将此配置更改为以下内容：

```
server server1 host1:7054 check backup
server server2 host2:7054 check backup
server server3 host3:7054 check
server server4 host4:7054 check
```

用如下命令重启新配置的 HA 代理：

```
haproxy -f <configfile> -st $(pgrep haproxy)
```

"haProxyShowStats" 将会反映修改的配置，包括两个活动的旧版本备份服务端和两个升级的服务端（仍未启动）：

sid	pxname	svname	status	weig	act	bck
1	fabric-cas	server3	DOWN	1	1	0
2	fabric-cas	server4	DOWN	1	1	0
3	fabric-cas	server1	UP	1	0	1
4	fabric-cas	server2	UP	1	0	1

在 host3 和 host4 上安装 fabric-ca-server 升级版本的二进制文件。host3 和 host4 上新升级的服务端应该配置与 host1 和 host2 上较旧的服务端相同版本的数据库。启动升级的服务端后，数据库将自动迁移。haproxy 会将所有新流量转发给升级后的服务端，因为它们没有配置为备份的服务端。在继续其他操作之前，使用 "fabric-ca-client getcacert" 命令验证集群是否仍然正常工作。此外，"haProxyShowStats" 现在应该反映所有的服务器都处于活动状态，其类似于以下内容：

sid	pxname	svname	status	weig	act	bck
1	fabric-cas	server3	UP	1	1	0
2	fabric-cas	server4	UP	1	1	0
3	fabric-cas	server1	UP	1	0	1
4	fabric-cas	server2	UP	1	0	1

停止 host1 和 host2 上的旧服务端。在继续其他操作之前，使用 "fabric-ca-client getcacert" 命令验证新集群是否仍然正常工作。然后从 haproxy 配置文件中删除旧的服务端备份配置，使其类似于以下内容：

```
server server3 host3:7054 check
server server4 host4:7054 check
```

用如下命令重启新配置的 HA 代理：

```
haproxy -f <configfile> -st $(pgrep haproxy)
```

"haProxyShowStats" 将会反映修改的配置，包括两个已被升级为新版本的活动服务端：

sid	pxname	svname	status	weig	act	bck
1	fabric-cas	server3	UP	1	1	0
2	fabric-cas	server4	UP	1	1	0

10.3 fabric-ca-client

这一部分讲解如何使用 fabric-ca-client 的命令。

Fabric CA 客户端的根目录定义规则如下：

- ❑ 如果 -home 命令行参数已经设置，则使用该已设置的参数。
- ❑ 否则，如果存在环境变量 FABRIC_CA_CLIENT_HOME，则使用该环境变量。
- ❑ 否则，如果存在环境变量 FABRIC_CA_HOME，则使用该环境变量。
- ❑ 否则，如果存在环境变量 CA_CFG_PATH，则使用该环境变量。
- ❑ 否则，使用 \$HOME/.fabric-ca-client 作为根目录。

下面的指引假设客户端的配置文件存在于客户端根目录。

10.3.1 登录启动用户

首先，如果需要在配置文件自定义 CSR（证书签名请求），则 csr.cn 必须设置为引导身份的 ID。默认 CSR 如下：

```
csr:
  cn: <<enrollment ID>>
  key:
    algo: ecdsa
    size: 256
  names:
    - C: US
    ST: North Carolina
    L:
    O: Hyperledger Fabric
    OU: Fabric CA
  hosts:
    - <<hostname of the fabric-ca-client>>
  ca:
    pathlen:
    pathlenzero:
    expiry:
```

运行 fabric-ca-client enroll 命令来登录一个身份。比如，下面的命令向一个运行在本地 7054 端口的 Fabric CA 服务端登录了一个 ID 为 admin，密码为 adminpw 的身份。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client enroll -u http://admin:adminpw@localhost:7054
```

登录命令会存储一个登录证书（ECert）、相对应的私钥，还有 CA 证书链 PEM 文件。这些存储在 Fabric CA 客户端的 msp 目录的子目录下，你会看到信息提示 PEM 存储在哪里。

10.3.2 注册一个新身份

只有已经登录了的身份才能发起注册请求，而且必须有相应的权限才能注册相应身份类型。

注册时 Fabric CA 服务端要做六项权限检查：

(1) 注册者（即发起者）的“`hf.Registrar.Roles`”属性中必须有请求注册的类型。举个例子，如果发起者的“`hf.Registrar.Roles`”属性的值为“`peer,app,user`”，那么他能注册的类型为 `peer`、`app` 和 `user`，不能注册 `orderer`。

(2) 发起者的 `affiliation` 必须与他请求注册的身份的 `affiliation` 相同，或者是所请求的 `affiliation` 的前缀。举个例子，一个 `affiliation` 为“`a.b`”的发起者，可以注册一个 `affiliation` 为“`a.b.c`”的身份，但是不能注册一个 `affiliation` 为“`a.c`”的身份。如果一个身份需要根 `affiliation`，`affiliation` 应该为“`.`”，并且发起者必须具有根 `affiliation`。如果注册请求中没有指定 `affiliation`，正在注册的身份的 `affiliation` 将被赋予发起者的 `affiliation` 的值。

(3) 如果满足下述所有条件，注册者可以注册一个拥有属性的用户。

(4) 只有当注册者拥有该属性并且该属性是 `hf.Registrar.Attributes` 属性的一部分时，注册者可以注册具有‘`hf`’前缀的 Fabric CA 的保留属性。此外，如果属性值是列表类型，被注册的属性的值则必须是注册者拥有的属性值的子集。如果属性值是布尔类型，只有当注册者该属性的值为‘`true`’时，该属性才能被注册。

(5) 注册自定义属性（名称不以‘`hf.`’开头的任何属性）要求注册者拥有‘`hf.Registrar.Attributes`’属性，并且它的值为要注册属性的值或者样式。唯一支持的样式是末尾带有“`*`”的字符串。例如，“`a.b.*`”是符合所有名称以“`a.b.`”开头的属性的样式。例如，如果注册者的 `hf.Registrar.Attributes=orgAdmin`，则注册者可以从注册身份中添加或移除的属性即为‘`orgAdmin`’。

(6) 如果请求注册的属性名称是‘`hf.Registrar.Attributes`’，则需要额外的检查来查看该请求值是否等于注册者‘`hf.Registrar.Attributes`’属性的值或是其子集。为了满足这个条件，每一个请求的值都必须匹配注册者‘`hf.Registrar.Attributes`’属性的值。例如，如果注册者‘`hf.Registrar.Attributes`’属性的值为‘`a.b.*,x.y.z`’，请求属性的值为‘`a.b.c,x.y.z`’，则该请求是有效的。因为‘`a.b.c`’和注册者的‘`a.b.*`’匹配，而‘`x.y.z`’和注册者的‘`x.y.z`’匹配。

来看一个例子，有效场景如下：

(1) 如果注册者具有属性‘`hf.Registrar.Attributes = a.b.*, x.y.z`’，正在注册的属性为‘`a.b.c`’，则请求有效，因为‘`a.b.c`’匹配‘`a.b.*`’。

(2) 如果注册者具有属性‘`hf.Registrar.Attributes = a.b.*, x.y.z`’，正在注册的属性为‘`x.y.z`’，则请求有效，因为‘`x.y.z`’有效匹配注册者的‘`x.y.z`’。

(3) 如果注册者具有属性‘`hf.Registrar.Attributes = a.b.*, x.y.z`’，正在注册的属性为‘`a.b.c,x.y.z`’，则请求有效，因为‘`a.b.c`’匹配‘`a.b.*`’且‘`x.y.z`’匹配注册者的‘`x.y.z`’。

(4) 如果注册者具有属性‘`hf.Registrar.Roles = peer,client`’，且正在注册的属性值为‘`peer`’或者‘`peer,client`’，则请求有效，因为请求属性的值等于注册者属性的值或是其子集。

表 10-4 所示列出了能够被一个身份注册的所有属性（属性名称大小写敏感）。

表 10-4 身份属性表

Name	Type	Name	Type
hf.Registrar.Roles	List	hf.Revoker	Boolean
hf.Registrar.DelegateRoles	List	hf.AffiliationMgr	Boolean
hf.Registrar.Attributes	List	hf.IntermediateCA	Boolean
hf.GenCRL	Boolean		



注意 当注册一个身份时，你可以指定一组属性名称和值。如果组中多个元素具有相同名称，只有最后一个元素会被采用。也就是说，目前还不支持多值属性。

下述命令将使用管理员身份去注册一个新用户，其登录的 ID 为 “admin2”，affiliation 为 “org1.department1”，具有名为 “hf.Revoker” 值为 “true” 和名为 “admin”，值为 “true” 的属性。“:ecert” 后缀意味着默认情况下，“admin” 属性及其值将被插入到用户的登录证书中，然后可以用它来做出访问控制决定。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name admin2 --id.affiliation org1.department1
--id.attrs 'hf.Revoker=true,admin=true:ecert'
```

密码又被称为登录密码，登录身份时需要密码。所以，管理员可以注册一个身份，并将登录 ID 和密码给其他人使其登录该身份。

多个属性可以位于 `--id.attrs` 项下，且每个属性必须被逗号分隔。包含逗号的属性值，必须放在双引号中，如下所示：

```
fabric-ca-client register -d --id.name admin2 --id.affiliation org1.department1
--id.attrs '"hf.Registrar.Roles=peer,user",hf.Revoker=true'
```

或者

```
fabric-ca-client register -d --id.name admin2 --id.affiliation org1.department1
--id.attrs '"hf.Registrar.Roles=peer,user"' --id.attrs hf.Revoker=true
```

你可以通过编辑客户端的配置文件为注册命令中使用的各个项设置默认值。假设配置文件如下所述：

```
id:
  name:
  type: user
  affiliation: org1.department1
  maxenrollments: -1
  attributes:
    - name: hf.Revoker
      value: true
```



```
- name: anotherAttrName
  value: anotherAttrValue
```

下述命令将注册一个 ID 为 “admin3” 的身份。该身份从命令行获得，但剩余的配置则从配置文件获得，如身份类型为 “user”，affiliation 为 “org1.department1”，两个属性为 “hf.Revoker” 和 “anotherAttrName”。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name admin3
```

要注册一个具有多个属性的身份，需要像上所述般将所有属性名和值写入配置文件中。

将 maxenrollments 设置为 0 或将其从配置中删除，将导致身份被注册为使用 CA 的最大登录值。此外，注册身份的最大登录值不能超过 CA 的最大登录值。例如，如果 CA 的最大登录值为 5，则任何新身份的值必须小于或等于 5，当然也不能将其设置为 -1（无限登录）。

接下来，我们将注册一个节点身份，在接下来的章节我们将用该身份登录节点。下述命令注册了一个 peer1 身份。注意，我们自己选择规定自己的密码而不是让服务器为我们生成。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name peer1 --id.type peer --id.affiliation org1.
department1 --id.secret peer1pw
```

请注意，除非服务器配置文件中指定非叶隶属关系始终以小写形式存在，否则需要区分大小写。服务端 affiliations 部分的配置文件如下：

affiliations:

```
BU1:
  Department1:
    - Team1
BU2:
  - Department2
  - Department3
```

BU1、Department1、BU2 以小写形式存储。这是因为 Fabric CA 使用 Viper 读取配置。Viper 不区分映射键的大小写，始终返回小写值。要注册与 bu1.department1、Team1 关联的身份，Team1 需要在 -id.affiliation 项中指定，如下所示：

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client register --id.name client1 --id.type client --id.affiliation
bu1.department1.Team1
```

10.3.3 登录一个节点

现在你成功地注册了一个节点身份，你可以用 ID 和密码登录。这部分与登录一个引导身份类似。我们还会介绍如何使用 -M 选项来更换 MSP 的目录。

下面的命令用于登录 peer1。记得在 -M 选项下更改你自己的 MSP 目录，MSP 目录是由节点的 core.yaml 里的 mspConfigPath 指定的。你也可以设置 FABRIC_CA_CLIENT_

HOME 环境变量为 peer 的根目录。

```
# export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
# fabric-ca-client enroll -u http://peer1:peer1pw@localhost:7054 -M $FABRIC_CA_CLIENT_HOME/msp
```

登录一个 orderer 几乎是一样的，唯一不同的是 MSP 目录是设置在 orderer 的 orderer.yaml 文件里的 LocalMSPDir 中。

由 fabric-ca-server 颁发的所有登录证书具有以下组织单位（或简称为“OU”）：

- (1) OU 层次结构的根等于身份类型。
- (2) OU 被加到身份 affiliation 的每一个组件。

例如，一个节点类型的身份，它的 affiliation 为 department1.team1，那么该身份的 OU 层次结构（从叶到根）为 OU=team1, OU=department1, OU=peer。

10.3.4 从另一个 Fabric CA 服务器获得 CA 证书链

通常 MSP 目录的 CA 证书目录必须包含证书链，代表这个节点所有信任的信任中心。fabric-ca-client getcacerts 命令用于从其他 Fabric CA 服务器实例获取这些证书链。举个例子：下面的命令会在本地启动第二个 Fabric CA 服务器，监听 7055 端口，命名为 CA2。这代表两个由不同成员管理并且分开的信任中心。

```
export FABRIC_CA_SERVER_HOME=$HOME/ca2
fabric-ca-server start -b admin:ca2pw -p 7055 -n CA2
```

下面的命令会把 CA2 的证书链安装进 peer1 的 MSP 目录。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
fabric-ca-client getcacert -u http://localhost:7055 -M $FABRIC_CA_CLIENT_HOME/msp
```

默认情况下，Fabric CA 服务器会按照子级优先的顺序返回 CA 链。这意味着链中的每一个 CA 证书后面都有其颁发者的 CA 证书。如果你需要 Fabric CA 服务器按照相反的顺序返回 CA 链，就将环境变量 CA_CHAIN_PARENT_FIRST 设置为 true，然后重启 Fabric CA 服务器。Fabric CA 客户端将会合理处理每一种顺序。

10.3.5 重新登录一个身份

假设你的登录证书快过期了，你可以重新登录来替换你的登录证书（ECert）。

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/peer1
fabric-ca-client reenroll
```

10.3.6 撤销一个证书或身份

身份和证书都能被撤销。撤销一个身份会撤销该身份拥有的所有证书，该身份也不能再获得新的证书。撤销一个证书会使该证书失效。

为了撤销一个证书或身份，发起者必须有 `hf.Revoker` 和 `hf.Registrar.Roles` 属性。发起者只能撤销与自己 `affiliation` 相同的证书或身份，或者以其 `affiliation` 为前缀的证书或身份。此外，撤销者只能撤销在其 `hf.Registrar.Roles` 属性中列出的类型的身份。

举个例子，一个 `affiliation` 为 “orgs.org1”，并且具有 ‘`hf.Registrar.Roles=peer,client`’ 属性的撤销者只能撤销 `affiliation` 为 `orgs.org1` 或者 `orgs.org1.department1` 的 `peer` 或者 `client` 类型的身份，而不能撤销 `affiliation` 为 `orgs.org2` 或者其他类型的身份。

下面的命令撤销一个身份。将来所有发自该身份的请求都会被 Fabric CA 服务器拒收。

```
fabric-ca-client revoke -e <enrollment_id> -r <reason>
```

下面是 `-r` 选项支持的参数理由：

- (1) unspecified;
- (2) keycompromise;
- (3) cacompromise;
- (4) affiliationchange;
- (5) superseded;
- (6) cessationofoperation;
- (7) certificatehold;
- (8) removefromcrl;
- (9) privilegewithdrawn;
- (10) aacompromise.

举个例子，有着根 `affiliation` 的 `admin` 可以回收 `peer1` 身份：

```
export FABRIC_CA_CLIENT_HOME=$HOME/fabric-ca/clients/admin
fabric-ca-client revoke -e peer1
```

一个身份可以撤销自己的登录证书 (ECert)，需要指定 ECert 的 AKI 和序列号：

```
fabric-ca-client revoke -a xxx -s yyy -r <reason>
```

举个例子，你可以通过 `openssl` 命令来获取一个证书的 AKI 和序列号：

```
serial=$(openssl x509 -in userecert.pem -serial -noout | cut -d "=" -f 2)
aki=$(openssl x509 -in userecert.pem -text | awk '/keyid/ {gsub(/
    *keyid:|:\/,|"/, $1);print tolower($0)}')
fabric-ca-client revoke -s $serial -a $aki -r affiliationchange
```

`-gencrl` 选项可用于生成包含所有撤销证书的 CRL (证书吊销列表)。例如，以下命令将撤销身份 `peer1`，生成一个 CRL 并将其存储在 `<msp folder>/crls/crl.pem` 文件中。

```
fabric-ca-client revoke -e peer1 --gencrl
```

CRL 也可以使用 `gencrl` 命令生成。有关 `gencrl` 命令的更多信息，请参阅生成 CRL (证书撤销列表) 部分。



10.3.7 生成一个 CRL

在 Fabric CA 服务器撤销证书之后，Hyperledger Fabric 中相关的 MSPs 也应该被更新。这包含了 peer 节点本地的 MSPs channel 配置区块中相关的 MSPs。为此，PEM 编码的 CRL（证书撤销列表）文件必须放置在 MSP 的 crls 文件夹中。fabric-ca-client gencrl 命令可用于生成 CRL。任何具有 hf.GenCRL 属性的身份都可以创建一个 CRL，该 CRL 包含在特定时间段内被撤销的所有证书的序列号。创建的 CRL 存储在 \<msp folder> /crls/crl.pem 文件中。

以下命令将创建一个包含所有撤销证书（过期和未过期）的 CRL，并将其保存在 ~/msp/crls/crl.pem 文件。

```
export FABRIC_CA_CLIENT_HOME=~/.clientconfig
fabric-ca-client gencrl -M ~/msp
```

下述命令将创建一个包含 2017-09-13T16:39:57-08:00（通过 --revokedafter 选项指定）之后，2017-09-21T16:39:57-08:00（通过 --revokedbefore 选项指定）之前所有撤销证书（过期和未过期）的 CRL，并将其保存在 ~/msp/crls/crl.pem 文件。

```
export FABRIC_CA_CLIENT_HOME=~/.clientconfig
fabric-ca-client gencrl --caname "" --revokedafter 2017-09-13T16:39:57-08:00
--revokedbefore 2017-09-21T16:39:57-08:00 -M ~/msp
```

--caname 选项指定此项请求要发送的 CA 名称。在上述例子中，gencrl 请求将被发送给默认 CA。

--revokedafter 和 --revokedbefore 选项将指定时间段的下限和上限。生成的 CRL 将包含在此期间被撤销的证书。时间必须是 RFC3339 格式的 UTC 时间戳。--revokedafter 时间戳不能大于 --revokedbefore 时间戳。

默认情况下，CRL 的“下次更新”日期设置为第二天。crl.expiry CA 配置属性可以指定自定义值。

gencrl 命令还可以使用 -expire after 和 -expire before 选项，这些选项可用于生成在这些选项指定的时间段内过期的撤销证书的 CRL。例如，下述命令将生成一个包含 2017-09-13T16:39:57-08:00 之后且 2017-09-21T16:39:57-08:00 之前撤销的，2017-09-13T16:39:57-08:00 之后且 2017-09-21T16:39:57-08:00 之前过期的所有证书的 CRL。

```
export FABRIC_CA_CLIENT_HOME=~/.clientconfig
fabric-ca-client gencrl --caname "" --expireafter 2017-09-13T16:39:57-08:00
--expirebefore 2018-09-13T16:39:57-08:00 --revokedafter 2017-09-13T16:39:57-08:00
--revokedbefore 2017-09-21T16:39:57-08:00 -M ~/msp
```

10.3.8 启用 TLS

这一部分介绍如何为 Fabric CA 客户端配置 TLS。

下面的代码可以配置在 fabric-ca-client-config.yaml 中。



```

tls:
  # Enable TLS (default: false)
  enabled: true
  certfiles:
    - root.pem
  client:
    certfile: tls_client-cert.pem
    keyfile: tls_client-key.pem

```

certfiles 是该客户端信任的根证书集合。一般这都会是 Fabric CA 服务端根目录下的 ca-cert.pem。

只有在服务器配置了双向 TLS 的情况下，client 选项才需要。

10.3.9 基于属性的访问控制

访问控制决策可以通过 chaincode (Hyperledger Fabric 运行时) 根据身份的属性进行。这被称为基于属性的访问控制，简称为 ABAC。

为了实现这一点，一个身份的登录证书 (ECert) 可能包含一个或多个属性名称和值。然后通过链码提取属性的值以做出访问控制决定。

例如，假设您正在开发应用程序 app1，并希望特定的链码操作只能由 app1 管理员执行。你的链码可以验证调用者的证书 (由 CA 信任的 channel 颁发的证书)，它包含名为 app1Admin 且值为 true 的属性。当然，属性的名称可以是任何值，而且值不一定是布尔型的。

那么如何获得具有属性的登录证书？有两种方法：

方法一是当你注册身份时，可以指定一个登录证书并颁发给默认具有一个属性的身份。此行为可以在登录时被覆盖。这对于建立默认行为，假设注册发生在应用程序之外，且不需要任何应用程序更改的情况下很有用。

方法二是以下显示了如何用两个属性 app1Admin 和 email 注册 user1。当用户在登录时没有明确请求属性时，“: ecert” 后缀会使 appAdmin 属性默认插到 user1 的登录证书中。而电子邮件属性默认未添加到登录证书中。

```

fabric-ca-client register --id.name user1 --id.secret user1pw --id.type user
--id.affiliation org1 --id.attrs 'app1Admin=true:ecert,email=user1@gmail.com'

```

当你登录一个身份时，你可能会明确要求将一个或多个属性添加到证书中。对于每个请求的属性，你可以指定该属性是否可选。如果没有选择请求并且身份不具有该属性，则会发生错误。

以下显示了如何使用电子邮件属性且不使用 app1Admin 属性登录 user1，也可选用电话属性 (如果用户拥有电话属性)。

```

fabric-ca-client enroll -u http://user1:user1pw@localhost:7054 --enrollment.attrs
"email,phone:opt"

```



表 10-5 显示了为每个身份自动注册的三个属性。

表 10-5 身份属性表

Attribute Name	Attribute Value
hf.EnrollmentID	The enrollment ID of the identity
hf.Type	The type of the identity
hf.Affiliation	The affiliation of the identity

要将任意属性默认添加到证书中，你必须明确使用“:ecert”显式注册该属性。例如，下述注册身份 user1 如果登录时没有指定属性，则它的‘hf.Affiliation’属性将会被添加到登录证书中。注意，affiliation 的值(org1)必须和‘-id.affiliation’及‘-id.attrs’选项的值相同。

```
fabric-ca-client register --id.name user1 --id.secret user1pw --id.type user --id.affiliation org1 --id.attrs 'hf.Affiliation=org1:ecert'
```

获取更多链码 API 和基于属性的访问控制信息，请参考 <https://github.com/hyperledger/fabric/tree/release/core/chaincode/lib/cid/README.md>。

10.3.10 动态更新服务器配置

本节将介绍在不重新启动服务器的情况下 fabric-ca-client 如何动态更新 fabric-ca-server 配置。本节中的所有命令都要求你首先执行 fabric-ca-client 登录命令进行登录。

1. 动态更新身份

本节将介绍如何使用 fabric-ca-client 动态更新身份。

如果客户端身份不满足以下所有条件，则会发生授权失败：

- ❑ 客户端必须拥有值列表用逗号分隔的“hf.Registrar.Roles”属性，且正在更新的身型的类型的值等于其中一个。例如，如果客户端身份具有值为“client,peer”的“hf.Registrar.Roles”属性，则客户端可以更新的身份类型为“client”或“peer”，而不是“orderer”。
- ❑ 客户端身份的 affiliation 必须等于要更新的身型的 affiliation，或者是要更新身份 affiliation 的前缀。例如，一个客户端的 affiliation 为“a.b”，那么它可以更新 affiliation 为“a.b.c”的身份，不可以更新 affiliation 为“a.c”的身份。如果一个身份需要根 affiliation，那么更新时请求需要指明 affiliation 为“.”，并且客户端必须具有根 affiliation。

下述内容介绍了如何添加、修改和删除一个 affiliation。

2. 获得身份信息

只要发起者满足上述强调的授权要求，发起者就可以从 fabric-ca 服务端检索有关身份



的信息。以下命令显示如何获取身份。

```
fabric-ca-client identity list --id user1
```

发起者也可以通过以下命令来请求检索其被授权查看的所有身份的信息。

```
fabric-ca-client identity list
```

1) 增加一个身份

下面将添加一个名为 ‘user1’ 的新身份。添加一个新身份需要执行与 ‘fabric-ca-client register’ 命令注册身份相同的操作。这里有两种方法可用于添加新身份。第一种方法是通 -json 选项，你可以用一个 JSON 字符串描述身份。

```
fabric-ca-client identity add user1 --json '{"secret": "user1pw", "type":  
  "user", "affiliation": "org1", "max_enrollments": 1, "attrs": [{"name": "hf.  
  Revoker", "value": "true"}]}'
```

下述命令添加一个具有根 affiliation 的用户。注意，affiliation 为 ‘.’ 意味着是根 affiliation。

```
fabric-ca-client identity add user1 --json '{"secret": "user1pw", "type": "user",  
  "affiliation": ".", "max_enrollments": 1, "attrs": [{"name": "hf.Revoker",  
  "value": "true"}]}'
```

第二种方法为直接通过选项添加身份，下述示例添加了 ‘user1’。

```
fabric-ca-client identity add user1 --secret user1pw --type user --affiliation .  
  --maxenrollments 1 --attrs hf.Revoker=true
```

表 10-6 列出了身份的所有字段，并决定了它们是必需的还是可选的，还有它们可能具有的默认值。

表 10-6 身份字段表

Fields	Required	Default Value
ID	Yes	
Secret	No	
Affiliation	No	Caller's Affiliation
Type	No	client
Maxenrollments	No	0
Attributes	No	

2) 修改一个身份

有两种可以修改现有身份的方法。第一种方法是通过 -json 选项，你可以通过 JSON 字符串描述对身份的修改。你可以在一个请求中修改多项。任何未修改的身份元素都将保留其原始值。



注意：最大登录为“-2”即表示使用 CA 最大登录设置。

下述命令将通过 `-json` 选项对一个身份进行多项修改。

```
fabric-ca-client identity modify user1 --json '{"secret": "newPassword", "affiliation":
  ".", "attrs": [{"name": "hf.Registrar.Roles", "value": "peer,client"}, {"name": "hf.
  Revoker", "value": "true"}]}'
```

第二种方法是利用下述命令将直接通过选项对身份进行修改。它将身份 `user1` 的登录密码修改为 `newsecret`。

```
fabric-ca-client identity modify user1 --secret newsecret
```

下述命令将身份 `user1` 的 `affiliation` 修改为 `org2`。

```
fabric-ca-client identity modify user1 --affiliation org2
```

下述命令将身份 `user1` 的类型修改为 `peer`。

```
fabric-ca-client identity modify user1 --type peer
```

下述命令将身份 `user1` 的最大登录修改为 5。

```
fabric-ca-client identity modify user1 --maxenrollments 5
```

下述命令将身份 `user1` 的最大登录修改为 -2，将使用 CA 的最大登录设置。

```
fabric-ca-client identity modify user1 --maxenrollments -2
```

以下内容将身份 `user1` 的 `hf.Revoker` 属性的值设置为 `false`。如果身份具有其他属性，它们不会更改。如果身份以前没有拥有 `hf.Revoker` 属性，则该属性将添加到身份中。通过为属性指定空值也可以删除一个属性。

```
fabric-ca-client identity modify user1 --attrs hf.Revoker=false
```

下述命令将属性 `hf.Revoker` 从身份 `user1` 中移除。

```
fabric-ca-client identity modify user1 --attrs hf.Revoker=
```

以下命令说明可以在单个 `fabric-ca-client` 身份修改命令中使用多个选项。在这种情况下，对用户 `user1` 的密码和类型进行修改。

```
fabric-ca-client identity modify user1 --secret newpass --type peer
```

3) 移除一个身份

下述命令将移除“`user1`”身份，并撤销与“`user1`”身份相关联的证书。

```
fabric-ca-client identity remove user1
```



注意 默认情况下，`fabric-ca-server` 禁用身份验证，但可以使用 `-cfg.identities.allowremove` 选项来启动 `fabric-ca-server`，以启用身份验证。

3. 动态更新 affiliation

本节介绍如何使用 fabric-ca-client 动态更新 affiliation。以下显示如何添加、修改、移除和列出 affiliation。

1) 添加一个 affiliation

如果客户端身份不满足以下所有条件，则会发生授权失败：

- ❑ 客户端身份必须具有属性 hf.AffiliationMgr，其值为 true。
- ❑ 客户端身份的 affiliation 层级必须高于所更新的 affiliation。例如，如果客户端的 affiliation 是 “a.b”，则客户端可以添加 affiliation “a.b.c” 而不是 “a” 或 “a.b”。

下述命令将添加名为 org1.dept1 的新 affiliation。

```
fabric-ca-client affiliation add org1.dept1
```

2) 修改一个 affiliation

如果客户端身份不满足以下所有条件，则会发生授权失败：

- ❑ 客户端身份必须具有属性 hf.AffiliationMgr，其值为 true。
- ❑ 客户端身份的 affiliation 层级必须高于所更新的 affiliation。例如，如果客户端的 affiliation 是 “a.b”，则客户端可以添加 affiliation “a.b.c” 而不是 “a” 或 “a.b”。
- ❑ 如果 “-force” 选项为真，并且有必须要修改的身份，则还必须授权客户端身份修改身份。

下述命令将 affiliation “org2” 重命名为 “org3”。同时也将重命名其子 affiliation（如 ‘org2.department1’ 将重命名为 ‘org3.department1’）。

```
fabric-ca-client affiliation modify org2 --name org3
```

如果重命名一个 affiliation 将会影响身份，不使用 -force 选项的话就会导致错误。使用 -force 选项将会更新这些受新的 affiliation 影响的身份的 affiliation。

```
fabric-ca-client affiliation modify org1 --name org2 --force
```

3) 移除一个 affiliation

如果客户端身份不满足以下所有条件，则会发生授权失败：

- ❑ 客户端身份必须具有属性 hf.AffiliationMgr，其值为 true。
- ❑ 客户端身份的 affiliation 层级必须高于所更新的 affiliation。例如，如果客户端的 affiliation 是 “a.b”，则客户端可以移除 affiliation “a.b.c” 而不是 “a” 或 “a.b”。
- ❑ 如果 -force 选项为真，并且有必须要修改的身份，则还必须授权客户端身份修改身份。

下述命令将移除 affiliation ‘org2’ 和它的子 affiliation。例如，如果 org2.dept1 是 org2 下的 affiliation，则它也将被移除。

```
fabric-ca-client affiliation remove org2
```



如果移除一个 affiliation 将会影响身份, 不使用 -force 选项的话就会导致错误。使用 -force 选项将会移除和这个 affiliation 相关的所有身份, 以及和这些身份相关的所有证书。

4) 列出 affiliation 信息

如果客户端身份不满足以下所有条件, 则会发生授权失败:

- ❑ 客户端身份必须具有属性 hf.AffiliationMgr, 其值为 true。
- ❑ 客户端身份的 affiliation 层级必须高于所更新的 affiliation。例如, 如果客户端的 affiliation 是 “a.b”, 则客户端可以获 affiliation “a.b.c” 或 “a.b” 的信息, 而不 “a” 或 “a.c”。

下述命令显示如何获取指定 affiliation。

```
fabric-ca-client affiliation list --affiliation org2.dept1
```

发起者也可以通过下述命令请求获取有权限查看的所有的 affiliation 的信息。

```
fabric-ca-client affiliation list
```

10.3.11 联系特定的 CA 实例

当服务器运行多个 CA 实例时, 可以直接请求到特定的 CA。默认情况下, 如果客户端请求中未指定 CA 名称, 则请求将被定向到 fabric-ca 服务器上的默认 CA。可以在客户端命令的命令行上指定 CA 名称, 如下所示:

```
fabric-ca-client enroll -u http://admin:adminpw@localhost:7054 --caname <caname>
```

10.4 HSM

默认情况下, Fabric CA 服务端和客户端将私钥存储在 PEM 编码文件中, 但也可以将它们通过 PKCS11 API 将私钥存储在 HSM (硬件安全模块) 中。此行为在服务端或客户端配置文件的 BCCSP (BlockChain 加密服务提供程序) 部分中配置。

配置 Fabric CA 服务端以使用 softhsm2

本节介绍如何配置 Fabric CA 服务端或客户端以使用 PKCS11 版本的 softhsm (请参阅 <https://github.com/openssl/openssl>)。

在安装 softhsm 之后, 创建一个令牌, 将其标记为 ForFabric, 将该引脚设置为 98765432。

你可以同时使用配置文件和环境变量来配置 BCCSP。例如, 按如下方式设置 Fabric CA 服务端配置文件的 BCCSP 部分。注意, 默认字段的值是 PKCS11。

```
#####
# BCCSP (BlockChain Crypto Service Provider) section is used to select which
# crypto library implementation to use
```

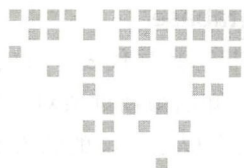


```
#####
bccsp:
  default: PKCS11
  pkcs11:
    Library: /usr/local/Cellar/softhsm/2.1.0/lib/softhsm/libsofthsm2.so
    Pin: 98765432
    Label: ForFabric
    hash: SHA2
    security: 256
    filekeystore:
      # The directory used for the software file-based keystore
      keystore: msp/keystore
```

你可以通过如下的环境变量来覆盖相关的字段：

```
FABRIC_CA_SERVER_BCCSP_DEFAULT = PKCS11
FABRIC_CA_SERVER_BCCSP_PKCS11_LIBRARY=/usr/local/Cellar/softhsm/2.1.0/lib/softhsm/
libsofthsm2.so
FABRIC_CA_SERVER_BCCSP_PKCS11_PIN = 98765432
FABRIC_CA_SERVER_BCCSP_PKCS11_LABEL = ForFabric
```





账本机制的设计与实现

本章主要讲解账本机制的设计与实现。

11.1 Ledger 架构概述

11.1.1 总览

1. 账本

账本是有序、无法篡改的所有状态变化的记录集合。状态变化是联盟链内参与的组织调用 chaincode（一次交易）的结果。每一次的交易都会导致一系列的键值对被创建、更新或删除，然后最新的状态信息被提交到账本中。账本本身就包含一个区块链用于存储无法篡改有序的数据记录，除了区块链之外还包含一个状态数据库用于维护当前的最新状态。每一个 channel 会维护一个账本。每一个 peer 也会保留一份它参与的 channel 的账本备份。

2. 链

一个链就是所有交易的记录，链的结构就是使用哈希作为指针的链表，其中链表的节点就是一个个区块，每一个区块包含多个交易。每个区块的头部包含了区块交易的哈希值，以及指向上一个区块头部的哈希值。这样一来，所有账本上的交易都以一种有序而且安全的方式被链接在一起。换句话说，在不破坏哈希链接的情况下，没有人能随意篡改存储在账本上的数据。链上最后一个区块的哈希值就代表了位于这个区块之前所有的交易，这样就可以保证所有维护该账本的 peer 都处在一个一致而且可信的状态。所有的链都被存储在 peer 的文件系统上，这样就可以高效地满足区块链数据只能追加而不能修改的特性。



3. State Database

账本当前的状态数据代表了链的交易记录中所有键值对的最新值，因为最新的状态信息就是 channel 已知的最新键值对，所以该状态信息也常常被称为世界状态（world state）。chaincode 的调用会根据当前的状态数据来执行交易。为了提高 chaincode 执行的效率，所有与键对应的最新的值都会被存储到状态数据库中。所谓的状态数据库也就是链中交易记录的索引视图，所以它随时可以从链中再重新生成一遍。当 peer 启动的时候，状态数据库就被恢复到当前最新的状态（如果有必要的话会重新生成一遍）。

4. 交易流程

我们先从一个比较高的层次来审视交易的流程：首先交易的提案从应用客户端发出，并发送给背书节点，背书节点对客户端的签名进行验证，如果验证成功则执行 chaincode 函数来模拟这次交易。这次模拟的输出就是 chaincode 的结果，分别是读集合和写集合。最后交易提案的回复与背书节点的签名一起被返回给客户端。客户端将所有的背书整合到一起形成一个交易负载并将其广播给一个排序服务。排序服务再将经过排序的交易形成区块发送给位于该 channel 中所有的 peer。在 peer 将区块提交到本地的 ledger 之前，它会检验该区块中所有的交易，首先根据背书策略来检测是否所有需要进行背书的节点都已经对交易结果进行签名，并且根据交易的负载验证签名的正确性。然后，peer 会对交易的读集合进行版本检测，以确保数据的完整性以及二次消费这样的危险。Hyperledger Fabric 为了增加整个系统的吞吐量，增加了交易在并行执行中并发控制的功能，并在区块被正式提交并写入区块链之前会保证所有被读的数据都没有经过修改。

5. 状态数据库的选择

我们可以选择 LevelDB 或者 CouchDB 作为我们的状态数据库：LevelDB 作为默认的键值数据库被嵌入到 peer 进程中。除了 LevelDB 之外，CouchDB 也可以成为可供选择的外置状态数据库，与 LevelDB 这种键值数据库类似，CouchDB 也可以存储二进制数据，但本身作为一个 JSON 文档存储。当数据都以 JSON 的格式存储时，我们就额外拥有了更加丰富的查询功能。LevelDB 和 CouchDB 都支持核心的 chaincode 操作，例如读取和设置一个键的值。查询键可以以范围的形式查询，也可以以组合键的形式进行查询。CouchDB 作为独立于 peer 之外的数据库进程而存在，所以使用 CouchDB 需要对设置、管理和操作有其他额外的考虑。刚开始你可以仅使用默认嵌入的 LevelDB，当你对查询开始有比较复杂的需求时，再将状态数据库迁移至 CouchDB。将 chaincode 的资产数据塑造成 JSON 的格式不失为一个不错的选择，因为如果选择 JSON 格式，你以后就可以进行复杂的查询操作了。

11.1.2 ledger 部分摘要

Ledger 是整个 Hyperledger Fabric 的核心组件之一，它封装了整个项目对区块链信息的存储、查询、索引等一系列的重要过程。首先我们先来了解一下 ledger 部分的目录结构，



ledger 部分主要存在于两个文件夹之下，分别是 `github.com/hyperledger/fabric/common/ledger` 与 `github.com/hyperledger/fabric/core/ledger`，我们可以从架构层面去理解为什么要将同一部分的代码分成两大部分，这样做可以实现代码的去耦合，也就是解耦的过程。core 部分代码直接可以为 common 部分的 ledger 提供底层服务，而 common 部分则可以为其他组件提供服务，层层封装，逻辑更加清晰。

1. common

下面是 ledger 在 common 部分的代码结构。

```

|— blkstorage
|   |— blockstorage.go
|   |— fsblkstorage
|       |— block_serialization.go
|       |— block_serialization_test.go
|       |— block_stream.go
|       |— block_stream_test.go
|       |— blockfile_helper.go
|       |— blockfile_helper_test.go
|       |— blockfile_mgr.go
|       |— blockfile_mgr_test.go
|       |— blockfile_rw.go
|       |— blockfile_scan_test.go
|       |— blockindex.go
|       |— blockindex_test.go
|       |— blocks_itr.go
|       |— blocks_itr_test.go
|       |— config.go
|       |— fs_blockstore.go
|       |— fs_blockstore_provider.go
|       |— fs_blockstore_provider_test.go
|       |— fs_blockstore_test.go
|       |— pkg_test.go
|— ledger_interface.go

```

通过上面的目录结构我们基本可以确定，这个部分要实现的核心功能就是区块的存储，这里还是以文件系统的方式存储区块。值得一提的是，golang 中接口思想贯穿了整个开源项目的主线，你可以把 `blkstorage` 看成接口，而子目录中的 `fsblkstorage` 则被看成对 `blkstorage` 的一种实现方式。虽然这里只有 fs 的实现方式，但是区块的存储方式绝对不仅限于此，比如你可以提供基于 RAM 的实现方式，提供 `ramblkstorage` 类似的实现。这只是一个例子，RAM 本身存在多种限制，所以不适合用于存储。这只是为了展示借口与实现相分离的思想。

下面我们还是从设计的角度出发，来理解各个文件存在的意义。假设你想要设计一个可以方便以特定格式存储，并且方便查询的区块存储系统，你会用到哪些东西呢？先从存储角度出发，你电脑的内存中拥有一些有效的区块信息，你要把它存入持久化的存储



中。首先，你肯定需要一种序列化的方式，即一种可以以固定的格式存入文件系统或从文件系统中读出的方法，这就是序列化的过程，`block_serialization.go` 就是为了实现相应的序列化功能而出现的。那拥有了将区块写入文件的方法之后，那还需要考虑一个问题，就是单个文件的大小。你会不会选择把所有的区块信息都写入一个文件呢？还是将区块写入不同的文件中？这些细节在我们设计软件时都是需要考虑的。正是这些微小的细节，它会决定我们软件是否高效和可靠，所以我们需要一个管理我们区块读写的模块，而 `blockfile_mgr.go` 正是用来管理如何存储区块的管理者。

区块文件查询，这是一个非常重要的组件，因为读取区块中的信息是非常常见的操作，如果设计得不恰当，它会直接成为整个系统的瓶颈。上面我们讨论过了，我们会将区块信息存入不同的文件中，那么，最终我们的文件系统中就会存在很多个区块文件，这样读写起来肯定不方便。因为如果要我们自己去负责每个文件的读写的话，工作量肯定会增大而且容易出错，所以我们可以将这些区块文件在逻辑层面上想象为区块的数据流。如果以“流”的思想去读取区块信息的话，操作起来就可以更加简便，这就是 `block_stream.go` 的作用。但是即使我们将这些区块文件想象成逻辑形式的流并没有解决我们如何读取各个信息单元的问题，所以 `blocks_itr.go` 就开始派上用场了。`itr` 就是 `iterator` 的意思，如果你对 C++ 里的 `iterator` 比较熟悉的话，那就非常好理解这个文件的作用了。它们的作用是类似的，就是以迭代器的形式读取信息。当然了，这种方式还是线形读取，在文件较多较大的情况下肯定是不适用的，所以我们就需要用到数据库中索引的思想，我们需要对区块文件建立索引，来加速搜索的功能，这就是 `blockindex.go` 的作用。以上就是 `common` 部分主要文件的功能，接下来我们来仔细研究一下文件中的具体细节。

在 `fsblkstorage/block_serialization.go` 中定义了区块信息的数据结构。

```
type serializedBlockInfo struct {
    blockHeader *common.BlockHeader
    txOffsets   []*txindexInfo
    metadata    *common.BlockMetadata
}

type txindexInfo struct {
    txID string
    loc  *locPointer
}
```

我们知道一个区块中包含了许多不同的交易，所以我们需要一个记录交易信息的数组，`[]*txindexInfo` 即可实现该功能，由它我们可以获取各个交易的序列号及它所在的位置。除了交易信息以外，一个区块还需要 `BlockHeader` 和 `BlockMetadata` 来维持区块的基本的信息，但是以上的数据结构仅是内存中的表示方式，我们需要一种能将这些信息持久化的方式将它们保存到硬盘（如数据库）中。序列化或者说编码的方式非常多种，从 `golang` 的标准库中我们就可以发现有 `JSON`、`XML` 等标准的编码方式，还可以使用 `gob` 这样 `golang` 独有的编码方式，选择一个编码方式需要考虑它的运行效率和得到结果的大小。



源码中使用的 google 的 protobuf 编码方式在速度和编码结果等方面都有不错的性能，所以项目中使用了它。

在设计 ledger 时，我们还需要一个管理器（blockfile manager）去管理用于区块持久化的文件，管理器可以管理区块文件系统中不同类型的信息，包括：用于存储文件的目录；用于存储不同区块的文件；用于记录最后被附加区块的文件的检查点；用于记录文件中不同区块和交易信息的索引。为此我们定义了如下的数据结构用于管理这些信息。

```
type blockfileMgr struct {
    rootDir      string
    conf         *Conf
    db           *leveldbhelper.DBHandle
    index        index
    cpInfo       *checkpointInfo
    cpInfoCond   *sync.Cond
    currentFileWriter *blockfileWriter
    bcInfo       atomic.Value
}
```

每当一个新的区块文件管理器被启动时，它会检测是否是自系统启动或重启后的第一次操作，若不是，则该操作被忽略。一个区块文件管理器将区块信息存入文件系统中，我们可以通过创建固定大小且序列号按顺序增长的文件来存储这些信息，例如 blockfile_000000、blockfile_000001 等。

下列步骤仅在系统启动时被执行：

- （1）检测用于存储信息的文件的文件夹是否存在，若不存在则创建文件夹。
- （2）检测键值数据库是否存在，若不存在则新建一个数据库（会创建一个 db 文件夹）。
- （3）确定用于存储的检查点信息的值（cpinfo），若存在则从 db 中载入，不存在则创建一个新的检查点（cpinfo）；如果检查点从 db 载入，则将其值与文件系统相比较；如果检查点与文件系统不同步，则将文件系统的信息同步入检查点中。
- （4）启动一个新的 file writer。

（5）确定用于在文件区块存储中查找交易和区块信息的索引。实例化一个新的 blockIdxInfo，若 db 中存在，则载入相应的索引信息。syncIndex 会把最后一个区块的索引与文件系统中的相比较。如果索引与文件系统不同步，则将索引同步至文件系统的状态。

上面我们了解了 Fabric 是如何存储区块信息的，还有一个非常重要的领域我们还没有涉及，那就是如何读取和查询区块的信息。因为我们之前提到过了，区块文件是以 blockfile_000000、blockfile_000001 这样的格式存储的，如果以单个文件形式读取区块信息固然可以成功，但这种方法并不自然，操作起来也非常麻烦。我们可以对这些文件进行进一步抽象，因为这些文件的作用都是相同的。我们在读取区块信息的时候可以把它们想象成一个数据流，以流的眼光去对待这些数据，会让我们的操作方便不少，这就是 block_stream.go 的作用。接下来我们看看在该文件中是如何将多个文件抽象成一个文件流的。



```

type blockfileStream struct {
    fileNum      int
    file         *os.File
    reader       *bufio.Reader
    currentOffset int64
}

type blockStream struct {
    rootDir      string
    currentFileNum int
    endFileNum    int
    currentFileStream *blockfileStream
}

```

这里定义了两个非常重要的数据结构，分别是 `blockfileStream` 和 `blockStream`。首先我们将单个文件抽象成一个数据流，`blockfileStream` 可以从单个文件中顺序读取区块信息，它从给定的偏移地址 (`currentOffset`) 开始读取一直读取到文件末尾。在基于 `blockfileStream` 的基础上，我们又定义了 `blockStream` 用于读取多个文件，`blockStream` 可以从多个文件中顺序读取区块信息，它从给定文件的偏移开始一直读到 `endFileNum` 的最后。这个数据结构里面定义了指向 `blockfileStream` 的指针 `currentFileStream` 用于真正底层的读取操作。到现在我们已经看到了非常多的类似的封装操作，其实写一个大型项目，做好 API 的封装是一件非常重要的事情，而且封装是否得当会直接影响项目的整体架构。

当然了，`block_stream.go` 也仅仅做到对文件抽象为流的操作，它并没有帮助我们遍历这些数据流并且取得我们想要信息的操作。`blocks_itr.go` 文件就在基于 `block_stream.go` 为我们提供遍历文件流得到相应信息，在 `blocks_itr.go` 中定义了 `blocks-Itr` 这个数据结构，利用它我们就可以完全基于文件流上的区块信息读取操作。

```

type blocksItr struct {
    mgr          *blockfileMgr
    maxBlockNumAvailable uint64
    blockNumToRetrieve  uint64
    stream         *blockStream
    closeMarker      bool
    closeMarkerLock   *sync.Mutex
}

```

以上就是 `ledger` 部分最重要的数据结构及其相应的操作。

2. core

以下部分是 `core` 目录下的文件夹结构，这里只列举了比较重要的部分。

```

|— customtx
|   |— custom_tx_processor.go
|— kvledger
|   |— custom_processor_test.go
|   |— example

```

```

|   |— history
|   |— kv_ledger.go
|   |— kv_ledger_provider.go
|   |— kv_ledger_provider_test.go
|   |— kv_ledger_test.go
|   |— marble_example
|   |— pkg_test.go
|   |— recovery.go
|   |— txmgmt
|— ledger_interface.go
|— ledgerconfig
|   |— ledger_config.go
|   |— ledger_config_test.go
|— ledgermgmt
|   |— ledger_mgmt.go
|   |— ledger_mgmt_test.go
|   |— ledger_mgmt_test_exports.go
|— ledgerstorage
|   |— pkg_test.go
|   |— store.go
|   |— store_test.go
|— pvtdatastorage
|   |— kv_encoding.go
|   |— store.go
|   |— store_impl.go
|   |— store_impl_test.go
|   |— test_exports.go
|— util
|   |— couchdb
|   |— txvalidationflags.go
|   |— txvalidationflags_test.go
|   |— util.go
|   |— util_test.go

```

看一遍目录结构，我们大致可以知道 ledger 的 core 部分拥有如下组件：ledgermgmt、ledgerconfig、ledgerstorage、pvtdatastorage 及 kvledger 等。首先我们来了解一下各个组件的主要作用。首先在实际应用中，我们可能不只拥有一个账本，不同的联盟之间会有不同的账本，而相同的联盟之间可能也会有多个账本，并且在实际存储过程中，我们通常是以文件形式来存储我们的账本的。而打开文件、关闭文件这些烦琐的操作，偏向管理的操作（因为一般的联盟链用户并不会用到这些操作），我们可以将它封装成管理组件，这就是 ledgermgmt 下的 ledger_mgmt.go 的作用。ledgerstorage/store.go 其实就是提供 common 中存储方案的一个接口（如使用文件系统存储）。在联盟链中，并不是所有的信息都是共享的，一些私有的信息我们并不想让盟友掌握，所以我们需要一些私有数据的存储机制，pvtdatastorage/store.go 就是为了提供私有信息存储机制而存在的。还有一个比较重要的组件——kvledger，kvledger 其实也是 ledger，它是对 ledger.Peer-Ledger 的实现，这种实现提供了一种键值数据模型，可以更加方便我们查找需要的信息。

以上就是 core 部分中 ledger 的主要内容。

下面我们分析几个比较重要的文件中的细节实现。在我们平时使用 Fabric 时最常用到的就是 YAML 配置文件了，在 ledgerconfig/ledger_config.go 中定义了比较多的常用的配置，以下是定义的部分函数名。

```
func IsCouchDBEnabled() bool
func GetRootPath() string
func GetLedgerProviderPath() string
func GetStateLevelDBPath() string
func GetHistoryLevelDBPath() string
func GetPvtWritesetStorePath() string
func GetBlockStorePath() string
func GetPvtdataStorePath() string
func GetMaxBlockfileSize() int
func GetQueryLimit() int
func GetMaxBatchUpdateSize() int
func IsHistoryDBEnabled() bool
func IsQueryReadsHashingEnabled() bool
func GetMaxDegreeQueryReadsHashing() uint32
```

因为 YAML 是一种静态文件类型，程序需要把这些配置读入内存中，Fabric 使用了 viper 来读取配置文件，这样我们就可以通过改变配置文件来改变程序运行时的状态，从而满足特定场景的需求。需要注意的是，因为我们是从外部的 YAML 文件读取配置，所以我们并不需要更改源代码，这就是一种解耦的过程，这为我们编程提供了极大方便。如果使用硬编码，每改变一次配置，我们就需要重新编译一次源文件，这种方式肯定是不妥当的。所以，有时候将需要考虑的问题从编译时推迟到运行时反而能提供不少便利。因为一个联盟链之间一般存在不只一个账本，在多个账本存在的情况下，我们需要一些数据结构记录我们打开的账本。要知道，一个账本从本质上来说就是一个文件，所以我们可以使用一个 map 来记录打开的账本。源码中定义了 var openedLedgers map[string]ledger.PeerLedger，在 ledgermgmt/ledger_mgmt.go 中还定义了诸多如打开、关闭等比较繁杂但却重要的功能，正是因为这些功能不是经常被客户端用到，所以使用一个 ledgermgmt 模块来管理这些繁杂的工作。在 ledgerstorage/store.go 中定义了两个重要的数据结构来实际提供区块数据和私有数据的存储功能。

```
type Provider struct {
    blkStoreProvider    blkstorage.BlockStoreProvider
    pvtdataStoreProvider pvtdatastorage.Provider
}

type Store struct {
    blkstorage.BlockStore
    pvtdataStore pvtdatastorage.Store
    rwlock       *sync.RWMutex
}
```

在这里我们可以看到源代码中使用了提供者模式 (Provider Pattern), 这种设计模式的用处也是为 API 提供定义与实现的分离, 我们发现这种设计思想在整个 Fabric 项目中甚至其他大型开源项目中都有体现, Provider 结构体中定义了两个接口, 我们可以根据需求来提供满足这些接口的实例。接口 BlockStore 用于存储和取回区块, 该接口的实现会将参数 IndexConfig 传入, 该参数的目的就是决定哪些东西可以通过索引在区块存储中可搜索到。

```
type BlockStore interface {
    AddBlock(block *common.Block) error
    GetBlockchainInfo() (*common.BlockchainInfo, error)
    RetrieveBlocks(startNum uint64) (ledger.ResultsIterator, error)
    RetrieveBlockByHash(blockHash []byte) (*common.Block, error)
    RetrieveBlockByNumber(blockNum uint64) (*common.Block, error) // blockNum of
        math.MaxUint64 will return last block
    RetrieveTxByID(txID string) (*common.Envelope, error)
    RetrieveTxByBlockNumTranNum(blockNum uint64, tranNum uint64) (*common.
        Envelope, error)
    RetrieveBlockByTxID(txID string) (*common.Block, error)
    RetrieveTxValidationCodeByTxID(txID string) (peer.TxValidationCode, error)
    Shutdown()
}
```

除了存储公开的区块信息以外, 我们可能还需要一种永久存储私有信息的机制, 因为私有信息应该与账本中的区块同步, 所以私有信息需要以原子操作的形式实现。为了实现这种功能, 接口 Store 的实现需要提供如 commit/rollback 的两阶段提交功能。具体的实现应如下: 首先, 私有数据通过 Prepare 函数提交给 Store; 然后区块会被附加到区块存储中; 最后根据区块写入成功与否, 分别调用 Commit 或者 Rollback 函数。Store 接口的实现应该能够在调用 Prepare 和 Commit/Rollback 函数之间容忍服务器宕机。

```
type Store interface {
    InitLastCommittedBlock(blockNum uint64) error
    GetPvtDataByBlockNum(blockNum uint64, filter ledger.PvtNsCollFilter)
        ([]*ledger.TxPvtData, error)
    Prepare(blockNum uint64, pvtData []*ledger.TxPvtData) error
    Commit() error
    Rollback() error
    IsEmpty() (bool, error)
    LastCommittedBlockHeight() (uint64, error)
    HasPendingBatch() (bool, error)
    Shutdown()
}
```

11.2 Ledger 之 Block-Storage

数据库分为 leveldb 和 couchDB 两种, 在 core.yaml 配置文件下的 ledger 区域中, state-Database 选项即为所选用的数据库, 默认选用 leveldb。leveldb 实现在 common/ledger/util

下, couchDB 实现在 core/ledger/util 下。这里的实现不是指数据库实现, 而是将涉及的第三方库的数据库对象包装进 Fabric 自己使用的一些对象 (如 common/ledger/util/leveldb-helper/leveldb_helper.go 中的 DB 对象) 之中。使用 couchDB, 可以支持更丰富的查询方式。

从用途角度讲, 数据库有用于存储账本 ID 的 idStore, 有用于存储 Block 的 BlockStore, 有用于存储链上交易当前最新状态的 VersionedDB, 有用于存储有效交易部分信息的 HistoryDB。

从使用者角度讲, peer 节点使用的账本的基本的结构为: 数据库 DB 被包含在一个账本实例 ledger 中, ledger 可以由一个账本提供者 ledger_provider 提供, ledger_provider 被包裹在一个账本管理者 ledger_mgmt 中, ledger_mgmt 供 peer 节点的其他模块直接使用。

从功能角度讲, 对于存储 block 的数据库, 由于区域链不允许对链上的数据进行修改和删除, 即已发生的数据永久不变地保留, 因此数据库有的增删改查操作, 到了 Fabric 中, 用于存储 block 的账本中只有增查两个操作, 加上数据库的开启和关闭两个操作, 剩余的就是 Fabric 账本针对区域链而保有的一些辅助功能。

11.2.1 peer 节点中的 leveldb

peer 节点使用 github.com/syndtr/goleveldb/leveldb 这个 leveldb 这个库, 其将 leveldb 的数据库对象和操作都封装进自己的对象中进行使用。在 common/ledger/util/leveldbhelper/ 下, 主要有三个对象:

(1) DB 在 leveldb_helper.go 中, 是一个数据库对象, 封装了配置 conf、leveldb、DB、同步 (或非同步) 的读写选项。

(2) DBHandle 在 leveldb_provider.go 中, 是一个数据库处理对象, 直接封装了 DB 和该数据库名称, 也是直接由账本使用的对象, 有 Put、Get、Delete 等基础操作。

(3) Iterator 在 leveldb_provider.go 中, 是数据库迭代器, 直接封装了 leveldb 的 Iterator, 也没有特别之处。

11.2.2 peer 节点中的账本

基础功能: 创建, 开启, 查, 增, 关闭。

辅助功能: 在各个账本章节中详述。

数据库: idStore, BlockStore, VersionedDB, HistoryDB。

peer 使用的账本的直接对象是 core/ledger/kvledger/kv_ledger.go 中的 kvLedger 对象, 除了 idStore 被更高层的 PeerLedgerProvider 管理外, 其余三个账本均由 kvLedger 持有并管理。

11.2.3 创建

使用账本前, 需要先创建账本对象。

peer channel join -b genesisblock.block 命令, 根据 genesisblock, 创建一个 channel。该命

令最终会触发以下过程。

```
Invoke(stub)-> joinChain(cid, block) (core/scc/cscs/configure.go)
```

cid 即为 channel 的 ID (这个 channel 的 ID 在后文实际上用作账本的 ID), block 即为 genesisblock。

```
peer.CreateChainFromBlock(block) (core/peer/peer.go)
-> l, err = ledgermgmt.CreateLedger(cb) (core/ledger/ledgermgmt/ledger_mgmt.go)
-> ledgerProvider.Create(genesisBlock) (core/ledger/kvledger/kv_ledger_provider.go)
-> ...
```

根据 genesisblock 创建账本并保存到了 ledger_mgmt.go 中的 openedLedgers 映射中。这一步账本被创建。

还是在 peer.CreateChainFromBlock(block) 中, 在创建完账本 l 后, 会执行 createChain(cid, l, cb) -> c := committer.NewLedgerCommitterReactive(ledger,...), 创建一个账本提交者对象 c, 而生成的账本 l 被 c 持有。账本提交者对象为 core/committer/committer_impl.go 中的 LedgerCommitter, 在 peer 中向账本提交数据或获取账本信息, 均是通过该对象的 Commit(block) 接口提交给账本的。committer.go 中接口的注释中的一句话可以诠释该提交者的目标: leave-everything-to-the-committer-for-now, 即现在把所有的事情交给 committer。

还是在 createChain(cid, l, cb) 中, 创建了账本提交者对象 c 后, 会执行:

```
service.GetGossipService().InitializeChannel(...,c,...) -> state.NewGossipState
Provider(...,committer,...) -> ...
```

c 传到 gossip 模块的 state 实例中, 被 state 持有。

这里主要针对第 3 点创建账本, 账本有创建和打开两个接口, 分别对应 core/ledger/kvledger/kv_ledger_provider.go 中的 Create(genesisBlock)、Open(ledgerID) 两个实现。创建包含了打开接口, 是在账本不存在时进行的操作, 创建的时候会将 genesisblock 直接写入账本。打开则是在账本已经存在时进行的操作。一个账本存在的标准: idStore 中存有该账本的 ID, 且该账本的在建标识不存在。

11.2.4 使用

在 peer 节点接收由 orderer 节点发送来的 block 数据后, 会交由 gossip 模块中的 state 子模块向账本添加 block。而添加 state 时, 则是通过账本提交者对象向账本添加 block 来实现的。具体过程如下, 在 gossip/state/state.go 中:

```
... -> deliverPayloads() -> s.commitBlock(rawBlock) -> s.committer.Commit(block)
```

这里的 committer 即为上一节创建后被传入 state 模块并由 state 模块持有的账本提交者对象 c。该文件其他函数中还会调用 committer.LedgerHeight(), 即利用账本提交者对象获取账本对象。

当 block 一路向账本提交，会到达 `core/ledger/kvledger/kv_ledger.go` 中的 `Commit(block)` 处，在这个函数中：

```
l.txtmgmt.ValidateAndPrepare(block, true)
```

验证并准备 block 数据，为向 VersionedDB 中写入做准备。

```
l.blockStore.AddBlock(block)
```

向 BlockStore 账本中写入 block。

```
l.txtmgmt.Commit()
```

在前面的准备下，这里将 block 数据写入 VersionedDB。

```
l.historyDB.Commit(block)
```

如果执行判断 `if ledgerconfig.IsHistoryDBEnabled()`，即 HistoryDB 配置使能，则也将 block 数据写入 HistoryDB 账本。

11.2.5 idStore

idStore 相对简单，对象为 `core/ledger/kvledger/kv_ledger_provider.go` 中的 `idStore`，直接使用了 `leveldb`。其主要有两个功能：存储创建的账本 ID；使用一个在建标识 (`ConstructionFlag`) 来标记账本是否正确建立，类似于一个检查点。

11.2.6 存储账本 ID

存储账本 ID 相当简单，只有增 / 查操作，以 “`ledgerKeyPrefix+ 账本 ID`” 为 key，以 `genesisblock` 为 value，组成键值对进行存储。对于 `ledgerIDExists`、`getAllLedgerIds` 两个接口，前者判断一个账本是否存在，后者获取所有建立成功的账本 ID。

11.2.7 ConstructionFlag

建立一个账本需要一定的时间，而若在建立账本过程中系统崩溃，则会出现只有部分账本建立的情况，`leveldb` 中可能残留部分账本数据，磁盘中可能残留部分账本文件，这些需要清除或修复，否则可能会影响账本的再次建立（实际上不影响，但是可以做，具体参看 `kv_ledger_provider.go` 中的 `runCleanup(...)`）。`idStore` 中以 `underConstructionLedgerKey` 为 key，账本 ID 为 value，组成键值对进行存储，从而作为一个在建标识，标记这个账本正在创建当中。在 `kv_ledger_provider.go` 中：

(1) `Create(genesisBlock)` 创建账本时，检查完该账本是否存在，若账本不存在，则立即 `provider.idStore.setUnderConstructionFlag(ledgerID)`，向 `idStore` 中添加在建标识，对当前创建的账本进行标记。

(2) `provider.openInternal(ledgerID)`，`ledger.Commit(genesis-`

Block), 先创建账本, 这需要一些列的操作, 然后再向账本中存储 genesisBlock。

(3) provider.idStore.createLedgerID(ledgerID, genesisBlock), 在这个函数中, batch.Put(key, val) 向 idStore 中存储该账本 ID, batch.Delete(underConstructionLedgerKey) 从 idStore 中删除在建标识。最后 s.db.WriteBatch(batch, true) 同步批量写入这两点改变, 如此这个账本才算完整建立。而在 WriteBatch 将在建标识真正从 idStore 中删除前任何时候系统发生崩溃, 下次系统重启时读取 idStore 时, 在建标识仍能被读取出来, 因此在建标识可以标识账本只被部分在建的情况, 此时需要执行 recoverUnderConstructionLedger 来恢复账本。

11.2.8 账本恢复

由 ConstructionFlag 标识最新的账本是否成功建立, 即 genesisBlock 是否成功写入。在创建账本管理对象 PeerLedgerProvider 时, 会主动尝试恢复账本 (参看 kv_ledger_provider.go 中的 NewProvider())。在 recoverUnderConstructionLedger() 中:

(1) ledgerID, err := provider.idStore.getUnderConstructionFlag() 获取当前 idStore 中的在建标识。若不存在, 则 ledgerID == "" 成立, 说明当前不存在不完整在建的账本, 程序返回; 否则会继续执行, 进行账本的恢复工作。

(2) ledger, err := provider.openInternal(ledgerID), 再次创建 VersionedDB, BlockStore, HistoryDB。bcInfo, err := ledger.GetBlockchainInfo(), 从 BlockStore 中读取 block 账本信息 (包括当前账本高度, 当前账本最新存储的 block 的哈希值, 上一块 block 的哈希值)。

(3) switch bcInfo.Height { ... }, 根据获取的账本高度, 分两种情况:
 ① case 0, 说明 genesisblock 未完整写入, 则执行 provider.runCleanup(ledgerID)、provider.idStore.unsetUnderConstructionFlag(), 即清理后删除在建标识。这说明两点, 一是 block 序列号确实是从 0 开始的, 二是在未写入 genesisblock 的情况下账本算作未建立。
 ② case 1, 说明 genesisblock 已写入, 则执行 genesisBlock, err := ledger.GetBlockByNumber(0), provider.idStore.createLedgerID(ledgerID, genesisBlock), 获取 genesisblock 块, 并据此在 idStore 中添加这个账本。



注意 这里的账本恢复是指在账本级别进行的恢复, 而每个账本在创建时因进行各种操作导致系统崩溃而造成的部分写入的情况, 都由账本自己负责恢复或清除。而恢复工作的重点是修复和同步数据, 而不是创建直接可以使用的账本对象 (创建可以使用的账本对象由其他接口负责, 如 kv_ledger_provider.go 中的 Create(genesisBlock)、Open(ledgerID)), 因此第 2 点所创建的各个账本对象, 只是为了让这些账本做一下自己所负责的恢复工作而已。这一点可以从第 2 点 bcInfo, err := ledger.GetBlockchainInfo() 之后就执行了 ledger.Close() 将创建的账本对象关闭的操作看出来。

11.2.9 BlockStore

BlockStore 是一个较底层的、基础的、公用的存储 block 块数据对象的接口，具体实现为 `common/ledger/blkstorage/fsblkstorage/blockfile_mgr.go` 中的 `blockfileMgr`，统筹管理 block 的存储操作。block 的存储状态信息和 block 数据自身是分开存储的：当前 block 块的存储状态存储在 `leveldb` 数据库中，block 块数据自身则存储在称为 `blockfile` 的文本文件中。换句话说，`leveldb` 用于存储当前账本的 block 数据的状态信息，这些信息主要指 block 的位置、长度信息、当前 `blockfile` 文件大小、`index` 信息等，主要用于定位 block 在 `blockfile` 中的位置，而 `blockfile` 文件保存实际的 block 数据。基本的操作方式也是根据 `leveldb` 中的信息去 `blockfile` 中对 block 进行读写操作，两者相互配合，共同保证 block 存储的完整性。另外有一个 block 迭代器对象，用于逐个读取 block 块数据的对象。

这里涉及几个对象（以下所说的 block 均指账本中当前存在的最新的 block，目录以 `common/ledger/blkstorage/fsblkstorage/` 为基准）。

1. blockfile

`blockfile` 用于存储 block 块数据的文件。具体实现为 `block_stream.go` 中的 `blockStream`，主要对 `blockfile` 进行读写操作。`blockfile` 文件名以 `blockfile` 为前缀，以六位数字为后缀，随着文件增加，这六位数字依次增加，如 `blockfile_000000`、`blockfile_000001`……

首先说一下 block 数据在 `blockfile` 中具体的存储方式，存储方式决定了具体如何读取和写入 block，所以应先讲。一个完整的 block 数据包包含两部分：block 数据长度信息 + block 数据自身，这里分别用 A 和 B 表示。block 数据包在 `blockfile` 中的偏移也是从 A 为开始算起的。如一块 block 数据为 `blockBytes`，则其长度是一个 `uint64` 类型的整数，即 `blockBytesLen = uint64(len(blockBytes))`，为了节约空间和方便读取，会用 `blockBytesEncodedLen := proto.EncodeVarint(blockBytesLen)` 对这个整数进行编码并形成变长的数据（原理是在大端系统中删除一个整数低位部分的 0）。这里的 `blockBytesEncodedLen` 就是 A，`blockBytes` 就是 B。然后 A 和 B 先后写入 `blockfile` 后，这块 block 数据就写入完成了。注意，A 是代表了 B 的长度的数据，A 本身也有个长度，即 `len(A)+len(B)` 才是这个 block 数据包的长度。其他的对象，比如保存点 `checkpointInfo`，会在写入的时候记录下该 block 在 `blockfile` 中的位置等信息。

从 `blockfile` 中读取 B 时，会直接在 block 数据包的开始处（也就是从 A 开始）直接读取 8 个字节（A 所在的位置由 `leveldb` 存储的 `checkpointInfo` 保存），然后调用 `length, n := proto.DecodeVarint(lenBytes)` 尝试解码 A 的实际值。这里直接读取 8 个字节，是假设这 8 个字节中一定会完整包含 A。另外，如当 A 只有 3 位，读取的 8 位数据的剩余 5 位为 B 的数据时，`DecodeVarint` 依然会解码出 A 来，此时返回的 `n` 为 3，`length` 则为 A 代表的实际值（也就是 B 的实际大小）。然后，根据解码出来的 `length`，以及 A 所占的 `n` 个字节，则可以定位到 B 开始的地方，然后读取 `length` 个字节，也就完整地把 B 读取出来了。

对于 B 是否完整, 是通过比较当前 blockfile 的大小与 $\text{length} + n + \text{checkpointInfo}$ 记录的 block 数据包的偏移值之和的值来判定的, 若后者小于当前 blockfile 值的实际大小, 则 B 自然是完整的。

读取 block 数据包, 是由 `block_stream.go` 中的 `blockStream` 和 `blockfileStream` 对象完成的。其中能体现上述读取过程的, 是两个对象的 `nextBlockBytesAndPlacementInfo` 接口。该接口每次从指定的文件中的指定位置开始, 尝试读取一个 block 数据包。

2. block 信息数据库

存储当前账本中 block 存储状态信息的 `leveldb` 数据库, 具体到 `blockfileMgr` 对象中的 `db`、`index` 两个成员 (两者实际上使用的是同一个 `leveldb` 数据库), 主要存储两种数据:

(1) 检查点 (checkpoint), 具体为 `blockfile_mgr.go` 中的 `checkpointInfo` 对象。每在 `blockfile` 中存储一个 block 数据包, 都会在 `leveldb` 中存储与该 block 数据包对应的检查点。检查点以 `blkMgrInfoKey` 为 key (`blockfile_mgr.go` 中), `checkpointInfo` 为 value, 从而组成一个键值对并存储在 `leveldb` 中。`latestFileChunkSuffixNum` 记录 block 所在的 `blockfile` 的文件名后缀, `latestFileChunksize` 记录当前 `blockfile` 最新的大小, `isChainEmpty` 标识当前账本是否为空, `lastBlockNumber` 记录账本当前最新 block 的序列号。

(2) 索引 (index), 具体为 `blockindex.go` 中的 `blockIndex` 对象。每在 `blockfile` 中存储一个 block 数据包, 都会在 `leveldb` 中存储与该 block 数据包对应的一批索引。索引分别以 block 序列号、block 哈希值、交易 ID 等为 key, 以账本中最新 block 的位置信息等为 value, 组成一批键值对, 并存储在 `leveldb` 中。这些索引项的 key 值预定义在 `common/ledger/blkstorage/blockstorage.go` 中, 主要是为调用者提供多种索引方式以在 `blockfile` 中定位 block 块数据, 比如: 有的想用 block 的序列号去查找一个 block 数据; 有的想使用 block 数据的哈希值去查找 block 数据; 有的想使用交易 ID 去查找一个 block 中的具体交易的数据等。

与 block 数据包对应的每批索引中, 包含了 block 块中每笔具体交易数据的索引, 具体存储在 `blockIndex` 对象的 `txOffsets` 中, 这个成员在计算每笔交易在 `blockfile` 中的偏移位置时, 随着添加 A+B, 总共经历了三轮: 首先计算的是每笔交易在 B 中的相对偏移; 然后计算的是每笔交易在 A+B 中的相对偏移; 最后计算的是每笔交易在 `blockfile` 中的偏移。索引自身也有一个检查点 `indexcheckpoint`, 用于记录当前最新的 block 数据包的索引的存储情况。

在写 block 数据包时, 当先后写入 A 和 B 后:

(1) 创建一个新的 `checkpointInfo` 对象 `newCPInfo`, 然后根据现有的 `checkpointInfo` 对象 `currentCPInfo` 和写入的 block 数据包填充 `newCPInfo`, 再非同步调用 `db.Put(...)`, 将新的 `checkpointInfo` 写入 `leveldb` 中, 覆盖掉 `currentCPInfo`。检查点的作用主要在于记录当前账本的保存状态, 也就是说记录账本当前保存到哪儿了、保存到哪个 block 了。

(2) 调用 `indexBlock(...)`, 进而调用 `batch.Put(...)`, 以批量写入的形式将与各个索引对应的信息写入 `leveldb` 中, 因为每批索引所使用的 `key` 不会一样, 因此不会发生覆盖。索引检查点会在每批索引的最后写入, 也即当索引检查点存在, 则当前这批索引一定存在。索引的作用主要在于为调用者提供多种查找具体 `block` 或交易数据的方式。

在此总结一下, 写入一个 `block`, 从前到后所涉及的每个数据如下:

```
block长度信息
block块数据
blkMgrInfoKey - checkpointInfo
blockHashIdxKeyPrefix+block哈希值 - block数据包位置信息
blockNumIdxKeyPrefix+blockID - block数据包位置信息
txIDIdxKeyPrefix+交易ID - 交易数据位置信息 (每个block中所含的所有交易)
blockNumTranNumIdxKeyPrefix+blockID+交易Seq - 交易数据位置信息 (每个block中所含的所有交易)
blockTxIDIdxKeyPrefix+交易ID - block数据包位置信息 (每个block中所含的所有交易)
txValidationResultIdxKeyPrefix+交易ID - 交易验证结果 (每个block中所含的所有交易)
indexCheckpointKey - block序列号
```

上述列表中, 可划分出两个范畴, 三个小整体: 1 和 2 存储到 `blockfile` 文件中, 属于一个范畴, 是一个小整体; 3~9 存储在 `leveldb` 中, 均为键值对 (`key-value pair`), 属于一个范畴; 3 为检查点键值对, 是一个小整体; 4~10 为索引键值对, 10 是索引检查点, 是一个小整体。`blockID` 指 `block` 的序列号, 交易 `ID` 指交易的序列号, 交易 `Seq` 指交易在 `block` 的 `Data` 数组中的下标序号。所有的 `key` 的...`Prefix` 前缀均在 `blockindex.go` 中定义。3 和 10 的每次更新都会覆盖旧的值。

`block` 数据包或交易数据的位置信息是一个 `blockindex.go` 中的 `fileLocPointer` 对象。该对象中, `fileSuffixNum` 记录数据所在的 `blockfile` 文件后缀, `offset` 记录数据在 `blockfile` 文件中的偏移, 也就是数据是从哪个位置开始的, `bytesLength` 记录数据的大小 (这个字段中, `block` 数据包未使用, 因为其长度存放在 `A` 中, 没必要使用这个字段。而交易数据使用了这个字段, 来记录每个交易数据的大小)。

同时, 正因为写入一个 `block` 时涉及多个、多类数据, 且为先后写入, 因此存在当系统崩溃时数据部分写入和三个小整体数据不同步的情况。因此在重新启动系统建立新的 `blockfileMgr` 时, 需要对部分写入的数据进行修补和对三个小整体的数据进行同步。同步时, 遵循一个隐含的规则是前面的数据未写入, 则后边的数据肯定没有写入。存在的情况有:

情况 1: 1 未完整写入。

情况 2: 1 完整写入, 2 未完整写入。

情况 3: 1~2 完整写入, 3 未完整写入。

情况 4: 1~3 完整写入, 4~10 未批量写入。

据此, 同步的步骤如下:

(1) 从 `leveldb` 中获取存有的检查点信息 `cpInfo` (如果没有则说明是新建的账本)。

(2) 根据 cpInfo 中存储的 block 数据包的位置信息, 定位该 block 数据包在哪个 blockfile 中的哪个位置。然后以此为开始尝试读取一个完整的 block 数据包, 对 cpInfo 进行更新同步。

(3) 这里需要辨别情况 1~3, 这三种情况下, 取出的 cpInfo 其实是上一个与 block 数据包对应的 cpInfo, 定位读取的时候也会完整读取出上一个 block 数据包, 然后再读取到此 block 时。当是情况为 1 和 2 时, 由于当前 block 数据包中的数据未完整写入, 因此会直接从上一个 block 数据包的末尾处截断 blockfile 文件, 删除当前 block 数据包不完整的数据, 此时也不需要再更新 cpInfo; 当是情况 3 时, 由于当前 block 数据包数据已完整写入, 因此会再读取当前 block 数据包, 依据当前 block 数据包的信息, 更新 cpInfo 和索引 (索引会使用更新过后的 cpInfo 进行更新)。当是情况 4 时, 取出的 cpInfo 为与当前 block 数据包对应的 cpInfo。cpInfo 不用更新, 只需要对索引信息进行同步。索引同步时, 会读取索引的检查点信息, 该索引检查点必然是上一个 block 数据包的索引检查点, 而由此获取的 block 数据包位置信息也是上一块的 block 的。跳过该 block, 之后的流程就同 cpInfo 更新的方式类似了。

3. block 迭代器

迭代器是调用者给一个开始的参数, 然后调用 Next() 依次遍历。当所要读取的 block 序列号大于账本中最新的 block 序列号时, 迭代器会等待, 直到新的 block 块产生并成功写入 blockfile 文件中。

4. 创建 - 增 - 查

在代码中追溯一下 BlockStore 的主要操作。以下涉及的代码的基准目录为 common/ledger/blkstorage/fsblkstorage/。

(1) 创建

1) 在 blockfile_mgr.go 中, newBlockfileMgr(...) 创建了一个新的 block 存储管理者, 接收了配置、索引配置和一个 leveldb 数据库三个参数, 其中配置中包含了 block 存储路径等信息。leveldb 以参数的形式传入, 给了更高层的管理对象 (如 fs_blockstore.go 中的 fsBlockStore、fs_blockstore_provider.go 中的 FsBlockstoreProvider) 更大的灵活性。

2) cpInfo, err := mgr.loadCurrentInfo(), 即是从 leveldb 中取出最新的检查点数据。当 if cpInfo == nil, 则说明当前新建的 block 存储管理者管理的是一个比较新的账本, 要么什么数据都未写入, 要么写入了第一块 block 数据包的部分数据, 此时可以将 cpInfo 更新为 blockfile 文件开始处的信息 (以供后续步骤尝试读取现有的 block 数据, 然后更新 cpInfo)。

3) syncCPInfoFromFS(rootDir, cpInfo), 是根据取出来的 cpInfo 同步更新的函数。在这个函数中, scanForLastCompleteBlock(...) 是尝试读取 blockfile 最

后一块 block 的函数，所给的三个参数分别表示哪个目录、哪个文件（后缀）、从文件里的哪个位置开始读，第三个参数赋的值为 `int64(cpInfo.latestFileChunksize)`，即 `cpInfo` 所对应的 block 数据包写入后文件的大小，也就是从 `cpInfo` 所代表的 block 数据包末尾的下一个位置（即下一个 block 数据包开始的位置）开始尝试读取一个完整的 block 数据包。如此，至多读取出一个完整的 block。读取的方法依然如上文所述，具体的实现由 `block_stream.go` 中的 `nextBlockBytesAndPlacementInfo()` 完成。若读取了这个 block 数据包，则会更新 `cpInfo` 的 `latestFileChunksize`（指向 blockfile 中最新的 block 数据包尾）、`lastBlockNumber`（更新为最新 block 的序列号）、`isChainEmpty`（更新为账本非空）三个字段。

4) `currentFileWriter.truncateFile(cpInfo.latestFileChunksize)`，根据更新后的 `cpInfo` 所存储的最后一个 block 数据包的末尾位置，直接截断该 blockfile 文件。这比较好理解，更新后的 `cpInfo` 的 `latestFileChunksize` 值之后的数据都是不完整的，所以直接截断，清除不完整的数据。

5) `mgr.syncIndex()`，根据更新过后的 `cpInfo` 和自身的检查点信息，创建同步索引。在这个函数中，`lastBlockIndexed, err = mgr.index.getLastBlockIndexed()` 即为获取索引的检查点，获取的是解码后的 block 序列号，也表示序列号为 `lastBlockIndexed` 的 block 数据包的索引已经完整存储。同样，若未获取，则说明所处理的账本较新，还没有索引存储进 `leveldb` 中。`flp, err = mgr.index.getBlockLocByBlockNum(lastBlockIndexed)` 是使用 `blockNumIdxKeyPrefix+blockID` 的索引方式获取序号为 `lastBlockIndexed` 的 block 数据包位置信息 `flp`。又因为能成功获取索引的检查点，所以可知该 block 数据包一定是完整存储的，因此可以直接跳过这个 block 数据包而去尝试读取下一个 block，这也是函数中 `skipFirstBlock` 变量所起到的作用。而一旦能成功读取下一个 block 数据包，则会调用 `info, err := extractSerializedBlockInfo(blockBytes)`，以根据读取出来的 block 数据包创建新的索引信息，然后调用 `mgr.index.indexBlock(blockIdxInfo)` 将新的 block 数据包的索引写入 `leveldb` 中，从而完成索引的更新同步。

(2) 增

1) `addBlock(block)` 对一个 block 数据块进行了添加。在这个函数中，`if block.Header.Number != mgr.getBlockchainInfo().Height` 首先验证了添加 block 数据的序列号是否是下一个 block 应有的，即序列号是否是连续的。`blockBytes, info, err := serializeBlock(block)` 将 block 数据进行串行化，以便转为可以直接写入 blockfile 文件中的数据，这里的 `blockBytes` 即为 B。在这里第一次计算了每笔交易数据在 block 块内的偏移位置（即以 `block.Data` 为开始计算每笔交易的偏移）。`currentOffset := mgr.cpInfo.latestFileChunksize` 即为当前 blockfile 的大小，也就是准备开始写当前添加的 block 数据包的位置。`blockBytesLen := len(blockBytes)`，`blockBytesEncodedLen := proto.EncodeVarint(uint64(blockBytesLen))` 压缩了 block 的大小值，这里

的 `blockBytesEncodedLen` 即为 `A`。`totalBytesToAppend := blockBytesLen + len(blockBytesEncodedLen)` 则为 `block` 数据包的总大小。如果执行判断 `if currentOffset+totalBytesToAppend > mgr.conf.maxBlockfileSize`, 即当前文件的大小与要添加的 `block` 数据包的大小之和大于配置中规定的 `blockfile` 的大小限制, 则会调用 `mgr.moveToNextFile()` 并直接启用下一个 `blockfile` 文件来存储这个新的 `block` 数据包。

2) `mgr.currentFileWriter.append(blockBytesEncodedLen, false)`, 非同步的在上一步确定的 `blockfile` 文件和文件中的位置处的写入 `A`, 第二个参数给的是 `false`, 因此此时 `A` 只是在缓存中而已。`mgr.currentFileWriter.append(blockBytes, true)`, 第二个参数是 `true`, 同步写入 `B`, 即与 `A` 一同写入 `blockfile` 磁盘文件中。需要注意的是, 虽然这里 `A` 和 `B` 是一起写入的, 但只是为了减少磁盘文件的操作。第二个参数为 `true`, 在 `append(...)` 函数 (`blockfile_rw.go` 中) 中会手动调用 `file.Sync()`, 即手工刷新缓存将缓存中的数据写入实际的磁盘文件。这个操作不是事务性的, 即在系统层面也是一个字符一个字符向磁盘文件中写的, 因此如果在写的过程中发生系统崩溃, 仍会出现 `A` 或 `B` 部分写入的情况。如果写入 `A` 和 `B` 的过程中出错, 则 `mgr.currentFileWriter.truncateFile(mgr.cpInfo.latestFileChunksize)` 直接把文件从最开始写的地方截断, 即将可能新写入的部分 `block` 数据包的数据删去, 然后添加程序结束并返回一个错误。

3) `newCPInfo := &checkpointInfo{...}`, 根据写入的 `block` 数据包和现有检查点的信息, 创建一个新的检查点。然后执行 `mgr.saveCurrentInfo(newCPInfo, false)`, 非同步地将新的检查点写入 `leveldb` 数据库中。同样, 若是检查点添加错误, 也是直接截断文件, 删除写入的 `block` 数据包, 将 `blockfile` 恢复如初, 结束添加并返回一个错误。

4) `blockFLP := &fileLocPointer{...}`, 根据新的检查点 `newCPInfo` 和开始写入 `block` 数据包时的偏移信息, 创建一个 `block` 数据包的位置信息 `blockFLP`。`for ... {txOffset.loc.offset += len(blockBytesEncodedLen)}`, 这里第二次计算 `block` 中包含的每笔交易的偏移, 即加上了 `A` 所占的字节数, 计算每个交易从 `A` 处开始算起的偏移量, 也可以说是每笔交易在 `block` 数据包内的偏移。`mgr.index.indexBlock(...)`, 将收集的索引用到的关于新的 `block` 数据包的信息传入, 创建一个新的 `block` 数据包的索引, 在这个函数中, 会逐个写入 1~6 的索引项, 其中在添加索引 3 和 4 (上文数据 6 和 7) 这两个与交易有关的索引时, 会调用 `txFlp := newFileLocationPointer(...)`, 以第三次计算每个交易的偏移信息, 即确定每笔交易在 `blockfile` 文件内的偏移, 同时也会在 `txFlp` 中记录每笔交易的大小。在所有索引项都批写入后, 然后通过 `batch.Put(indexCheckpointKey, encodeBlockNum(blockIdxInfo.blockNum))` 写入索引的检查点, 最后通过 `index.db.WriteBatch(batch, false)` 非同步写入 `leveldb` 数据库中。这里和第 3 点写入 `leveldb` 时都是用了非同步写入, 即执行后可能这些数据是写入 `leveldb` 的缓存中而非数据库中, 这样可以提高写入的效率。

5) `mgr.updateCheckpoint(newCPInfo)`, 这个主要更新了 `blockfileMgr` 对象保

存的检查点 `cpInfo`，然后调用了一个 `mgr.cpInfoCond.Broadcast()` 函数。这个函数主要使用 `sync.Cond` 的特性，用于通知 `block` 迭代器的 `Next()` 函数可能存在的等待，让其继续执行。上文已经说过，迭代器的 `Next()` 在遍历到序列号大于账本当前已有的最新 `block` 数据包的序列号时，会进入等待，一直到新的 `block` 数据包成功存储到 `blockfile` 中后，所以这里就在成功添加了一个 `block` 数据包之后进行通知。`mgr.updateBlockchainInfo(blockHash, block)`，这个函数使用 `atomic.Value` 的特性，原子性的存储链（也就是账本）的信息，即一个 `BlockchainInfo` 对象，该对象的 `Height` 属性保存了链的高度，`CurrentBlockHash` 保存了当前链存储的最新的 `block` 的哈希值，`PreviousBlockHash` 保存了上一个 `block` 的哈希值。

3. 查

查询 `block` 数据很简单，不同索引提供了不同的查询方式，均集中在 `blockfile_mgr.go`，是一系列 `retrieveBy` 格式的函数，名字很好理解，其中第一个空是你要查的东西；第二个空是你根据什么查这个东西，即通过什么来检索什么。这些函数主要通过 `leveldb` 中保存的 `block` 数据包的索引来获取 `block` 数据。

11.3 Ledger 之 VersionedDB

`VersionedDB` 账本又称状态账本，这里的状态（`state`），即是官方文档中所提及的世界状态（`world state`），即由每个交易数据最新生成的一个关于交易读写集的有效的键值对。该数据库中有几个重要的对象，如交易模拟器、查询器、验证器、读写集等。

`VersionedDB` 使用的 `state` 数据库有两个版本，即 `leveldb` 版本和 `couchdb` 版本，这里只讲 `leveldb` 版本。两者的主要区别在于：`leveldb` 只支持基本的以 `key` 为基础的查询；而 `couchdb` 则支持更丰富的查询手段，即富查询（参看 `core/ledger/util/couchdb/couchdb_test.go` 中的 `TestRichQuery`），对于调用者来说也这个版本更接近事物逻辑（如可以根据某账户的特征，如颜色、尺寸等进行查询，只需要预先定义好）。`state` 数据库也不是直接由账本使用的，而是被一个可称为交易管理者的对象 `TxMgr` 持有并管理，账本通过这个交易管理者向 `state` 数据库读写交易数据。同时，`TxMgr` 对象也提供其他功能，比如提供交易模拟器、交易查询器、交易验证器。这些“器”将在读写交易数据的时候发挥作用。`TxMgr` 的基础目录在 `core/ledger/kvledger/txmgmt/` 下，具体的实现为 `txmgr/lockbasedtxmgr/lockbased_txmgr.go` 中的 `LockBasedTxMgr`。

11.3.1 peer 节点使用 VersionedDB

在 `peer` 节点中使用 `VersionedDB` 的过程：用户使用 `peer` 节点发起一笔交易，如 `ACC` 的部署，是一个明显会写入数据的交易，再如用户执行 `peer chaincode query ...`，则是一个明显会读出数据的交易。在交易过程中，这些读取和写入的交易数据会

通过 TxSimulator (交易模拟器) 放入一个名为读写集的容器中。在交易返回时, 再使用 TxSimulator 统一读取, 然后将读取的数据放入 ProposalResponse 返回。返回后读集中的数据直接打印, 写集中的数据被放入 Envelope 中再发送给 orderer (请参看 peer/chaincode/common.go 中的 ChaincodeInvokeOrQuery)。orderer 节点依据 Envelope 生成 block 后, 再返回给 peer 节点。peer 节点接收数据后, 经 gossip 模块提交到账本对象, 账本对象添加 block, 会将 block 添加到 VersionedDB 账本中, VersionedDB 又会将 block 中交易的数据最终添加到 state 数据库中。以执行 chaincode_example02.go 的 invoke(stub,args) (该函数简单地查询了 A、B 两个账户的余额, 并由 A 向 B 转了一笔钱, 即形成了 A、B 两个账户转账调整后各自新的余额值) 为例, 步骤如下:

(1) 交易发起端执行 peer chaincode invoke -n chaincode_example02 -c '{"Args":["invoke","a","b","10"]}' , 通过 peer/chaincode/invoke.go 中的 chaincodeInvoke(...) 函数向 endorser 节点发送请求。此处假设该交易为 Tx_A。

(2) core/endorser/endorser.go 中, 通过 ProcessProposal -> txsim, err = e.getTxSimulator(chainID) 创建了一个交易模拟器。

(3) e.simulateProposal(...,txsim) -> e.callChaincode(...,txsim)。

(4) 经过一系列辗转, 会在 core/chaincode/shim/handler.go 的 handleTransaction 中调用 res := handler.cc.Invoke(stub) (即 chaincode_example02.go 中的 Invoke), 然后根据第 1 步的参数, 在 Invoke() 中进入 if function == "invoke" 分支从而调用 invoke(); 接着会经过一系列辗转, 会在 core/chaincode/handler.go 中的 enterBusyState, 通过交易模拟器的 txsimulator.GetState()、txsimulator.SetState(...), 将 A、B 账户的数据放入读集, 把更改的 A、B 账户的数据放入了写集 (其他交易会用了其他的 DeleteState(), GetStateRangeScanIterator(), ExecuteQuery()), 这一步相当于模拟执行了交易。

(5) 重回 core/endorser/endorser.go 中, simResult, err = txsim.GetTxSimulation-Results() -> 返回至 ProcessProposal()-> pResp, err = e.endorseProposal(..., simulationResult,...) -> e.callChaincode(...), 获取交易的写集, 并进行了第二轮的 callChaincode, 一系列调用后会进入下一步。

(6) 在 core/scc/escc/endorser_onevalidsignature.go 中, Invoke(...) -> presp, err := utils.CreateProposalResponse(...,results,...), results 为交易 Tx_A 的结果集, 其被放入 ProposalResponse 并返回至第 5 步的 endorseProposal, 在返回给交易发起端。

(7) 在 peer/chaincode/common.go 中, ChaincodeInvokeOrQuery() -> env, err := putils.CreateSignedTx(...), 把返回的 ProposalResponse 打包成 Envelope -> err = bc.Send(env), 发送给 orderer -> ..., orderer 处理 block 过程省略 -> orderer 将形成的 block 发送给 peer 节点, 再经 gossip 模块散播使用 committer 模块的过程, 这里亦省略。

(8) committer 向账本提交 block 时, 最终定位到 core/ledger/kvledger/kv_ledger.go 中的

`Commit(block) -> err = l.txmgtmgt.ValidateAndPrepare(block, true), l.txmgtmgt.Commit()` 即是使用交易管理者向 `state` 提交的交易数据。`ValidateAndPrepare` 做两件事，一是使用验证器验证交易的读写集，以确定交易的有效性；二是若交易有效，则将交易的写集中的数据放入数据升级包中，为下一步的 `l.txmgtmgt.Commit()` 提交这批数据做准备。

11.3.2 交易模拟器 / 交易查询器

交易模拟器 `TxSimulator` 和交易查询器 `QueryExecutor` 的接口在 `core/ledger/ledger_interface.go` 中定义，具体实现为 `txmgr/lockbasedtxmgr/` 下 `lockbased_tx_simulator.go` 中的 `lockBasedTxSimulator` 和 `lockbased_query_executor.go` 中的 `lockBasedQueryExecutor`。通过账本 `kvLedger` 的 `NewTxSimulator()`、`NewQueryExecutor()` 接口可获取 `TxSimulator` 和 `NewQueryExecutor`。两者均直接使用 `helper.go` 中的 `queryHelper` 实现了自身的查询方面的功能，这点从 `queryHelper` 这个名字可以看出来。`TxSimulator` 实际上包含 `QueryExecutor`，而 `QueryExecutor` 算是 `TxSimulator` 在查询功能上的增强和拓展（其中交易查询器的 `ExecuteQuery` 是富查询接口，因此只支持 `couchDB` 版本的 `VersionedDB`），因此直接使用的一般是交易模拟器。除了查询，`TxSimulator` 还提供状态的写入功能，写入分为增加和删除。因此，`TxSimulator` 和 `QueryExecutor` 涉及的操作就有查、增、删。

参看 `txmgr/lockbasedtxmgr/lockbased_tx_simulator.go` 和 `helper.go`，交易模拟器的读、写、删的操作步如下：

(1) `GetState(ns, key) -> q.helper.getState(ns, key)`，依据名字空间和 `key`，从 `state` 数据库中获取一个状态，返回并写入读集（返回的是值，写入读集的是值的版本）。这里的名字空间即 `chaincodeID`，也即对应一个 `chaincode` 的每一个交易使用一个读写集，下同。

(2) `SetState(ns, key, value) -> s.rwsetBuilder.AddToWriteSet(ns, key, value)`，将一个值写入写集。

(3) `DeleteState(ns, key) -> SetState(ns, key, nil)`，删除一个值，当给的 `key` 的 `value` 为 `nil` 时，即表示要将此 `key` 的值置为 `nil`，也即要删除这个值。

(4) `SetStateMultipleKeys(ns, kvs) -> for ... { SetState(...) }`，一次性设置多个键值对，写入交易的写集。

(5) `GetStateMultipleKeys(...)` -> `q.helper.getStateMultipleKeys(...)`，依据一个名字空间和一批 `key`，从 `state` 数据库中获取一批状态。

(6) `GetStateRangeScanIterator(...)` -> `q.helper.getStateRangeScanIterator(...)`，依据一个命名空间，根据开始的 `key` 和结束的 `key`，从 `state` 数据库中获取一个指定范围的交易查询迭代器。

(7) `ExecuteQuery(namespace, query) -> q.helper.executeQuery`

(namespace, query), leveldb 不支持这个接口。

(8) Done() -> q.helper.done(), 交易查询器执行完毕。

1. 读写集

TxSimulator 之所以叫交易模拟器, 就是因为在使用它处理交易的时候, 所形成的交易数据并未真正的直接写入 state 数据库中, 而是将交易查、增、删得到的数据暂时放入了一个名为读写集的地方备用, 因此是模拟交易。

一个 chaincode 的每笔交易都对应一个读写集, 读写集实现为 rwsetutil/rwset_builder.go 中的 RWSetBuilder, 成员只有一个 rwMap map[string]*nsRWs 映射, 即以 chaincodeID 为 key, 每个 chaincode 单独有一个 nsRWs。读写集存储三类数据: KVRead 读值、KVWrite 写值、RangeQueryInfo 范围读值 (三者均定义在 protos/ledger/rwset/kvrwset/ 下)。多个值各自形成 readMap 读集、writeMap 写集、rangeQueriesMap 和 rangeQueriesKeys 组成的范围读集。

读值: KVRead, 每个读值只包含 key 和值的版本号, 而不包含值本身。TxSimulator 每次调用 GetState() 接口, 都会将读取到的读值并将其写入交易的读集。

写值: KVWrite, 每个写值包含 key 和 value, 还包含一个决定此写值是否为删除的标识 IsDelete。TxSimulator 每次调用 SetState()、DeleteState() 等接口, 都会将一个写值写入交易的写集。

范围读值: RangeQueryInfo, 每个范围读值是一个范围内所有读值的集合, 范围读值何时被写入将在下文介绍。rangeQueriesMap 是以 rangeQueryKey 为 map 的 key, RangeQueryInfo 为 map 的 value。rangeQueryKey 中, startKey 标识了范围从何开始, endKey 标识了范围至何处结束 (注意, 范围读值中不包含与 endkey 对应的值), itrExhausted 标识了是否结束。RangeQueryInfo 中, 其余的成员同 rangeQueryKey 一样, 而成员 ReadsInfo 存储两类数据, 即原始的范围内的多个读值或范围内的读值的哈希值。范围内的多个原始的读值很容易理解, 而范围内读值的哈希值的生成则需要借助其他工具。

当用户发起读取一定范围数据的交易时, 不是直接将这个范围内所有读值集合打包返回给调用者, 而是将一个包含了读值范围信息的迭代器返回给调用者 (这样可以避免数据量过大而导致的处理效率低下), 然后调用者使用迭代器的 Next() 一个一个抽取读值。这里涉及两个迭代器:

(1) 在交易模拟器的范围内, 为 txmgr/lockbasedtxmgr/helper.go 中的 resultsItr, 由交易模拟器的 GetStateRangeScanIterator(...) 接口获取, 每获取一个迭代器, 都会把迭代器记录在 queryHelper 的 itr 中。再次强调, TxSimulator 包含 QueryExecutor, 而提供迭代器也是 QueryExecutor 对 TxSimulator 在查询功能上主要的拓展之一。

(2) 在核心的 chaincode 处理范围内, 为 core/chaincode/shim/chaincode.go 中的 StateQueryIterator。该迭代器的使用依附于 (1) 中迭代器所查询出来的结果。

2. 迭代器

resultsItr 迭代器管理了一个工具，即 rwsetutil/query_results_helper.go 中的 RangeQueryResultsHelper。RangeQueryResultsHelper 中的 pendingResults 用于暂时存储范围内的读值，merkleTree 用于存储一棵默克尔树，该树存储与 pendingResults 对应的哈希值，两者是随着数据的增加同步更新的（其实哈希值是否同步生成是由 hashingEnabled 决定，而该值又由账本的配置决定，该配置默认开启，由 core/ledger/ledgerconfig/ledger_config.go 中的 IsQueryReadsHashingEnabled() 接口决定）。每次 resultsItr 迭代器被调用一次 Next()，读取的读值都会添加到 RangeQueryResultsHelper 的 pendingResults 和 merkleTree 中暂存。

默克尔树也叫哈希树，实现为 query_results_helper.go 中的 merkleTree，由工具 RangeQueryResultsHelper 持有并管理。在 merkleTree 中，一个 map[MerkleTreeLevel][]Hash 格式的映射中实现了该树，树的每个节点保存的是哈希值。

默克尔树中有两个比较重要的值：Level 和 maxDegree。Level 标识树的层级，默认为 1，当树中的 Level1 有 maxDegree+1 个节点时，则会进行一个归并哈希值的操作：如两个节点存有 hash1、hash2，则该操作是将 hash1 和 hash2 的值前后连成一个整体，然后对这个整体进行哈希，得到一个新的哈希值 hash1-2，然后将 hash1-2 放入上一层（也就是 Level2）里面。下面也是举例子说明默克尔树的操作，对应 merkleTree 的 update(nextLeafLevelHash) 函数：

```
//设maxDegree==2，则连续插入1~9共9个哈希值，即调用9次merkleTree的update接口
//[x]-{hashN}，x为Level值，hashN代表N的哈希值。则tree的变化为：
```

1) [1]-{hash1}

2) [1]-{hash1,hash2}

3) [2]-{hash1-3}，插入第 3 个时，因为 len(currentLevelHashes) <= m.maxDegree 不成立，所有会向下一层，也就是第 2 层归集哈希值，同时删除第 1 层的值

4) [1]-{hash4},[2]-{hash1-3}

5) [1]-{hash4,hash5},[2]-{hash1-3}

6) [2]-{hash1-3,hash4-6}，插入第 hash6 时，同第 3 步原因一样和操作一样

7) [1]-{hash7},[2]-{hash1-3,hash4-6}

8) [1]-{hash7,hash8},[2]-{hash1-3,hash4-6}

9) [3]-{hash1-9}，这一步插入 hash9 时，hash7-9 向第 2 层归集，同时删除第 1 层的值，此时第 2 层也够了 3 个，此时会继续向第 3 层归集，归集第 2 层 hash1-3、hash4-6、hash7-9 三个哈希值，形成第 3 层的 hash1-9，同时会删除第 2 层的值

当工具 RangeQueryResultsHelper 执行 Done() 时，会将存储的 pendingResults 和 merkleTree 一同返回，两者实际上只会返回其中一个，其中一个必为 nil。当 pendingResults 中的存储读值的数量小于等于 maxDegree 时，是不会触发归并哈希值操作的，因此

此时返回的 `merkleTree` 为 `nil`。其他情况下, 因为 `hashingEnabled` 默认为 `true`, 所以只会返回一个归并后的唯一的哈希值。这个哈希值就代表了一个 `resultsIter` 迭代器调用 `Next()` 而被读出所有读值。返回一个归并后的哈希值而不返回原始的范围读集, 这样在效率上更高。另外从这一点还可以看出, 返回范围读值, 只可能是被用于验证的目的, 原因有二: 一是读值中本身就没有与 `key` 对应的值, 而只有版本号。二是这里可以只返回范围读集的哈希值。

以 `map.go` 为例, 当调用 `Invoke(stub)` 中的 `case "keys":` 分支时, `keysIter, err := stub.GetStateByRange(startKey, endKey)` 即获取了一个迭代器, 然后在 `for keysIter.HasNext()` 中依次抽取值。`keysIter` 即为 `StateQueryIterator`, 该迭代器中存储了执行范围查询操作后所查出来的读值集合:

(1) 在 `core/chaincode/handler.go` 的 `handleGetStateByRange(...)` 中, `rangeIter, err := txContext.txsimulator.GetStateRangeScanIterator(...)` 使用交易模拟器获取一个 `resultsIter`, 该迭代器会被放入 `queryHelper` 的 `itrs` 中。

(2) `payload, err = getQueryResponse(...)`, 使用 `resultsIter` 这个迭代器将指定范围内的结果查询出来, 放入 `payload`。这里就有一个 `maxResultLimit` 的限制, 即查询范围的起止最多是 100 个数据。同时也就是说, 传入的起止的 `key` 所得到的查询结果, 不一定是该范围内的全部数据。而指定范围内是否还有更多的数据, 会使用 `payload` 中的 `HasMore` 变量进行标识。

(3) `payload` 最终被返回到 `core/chaincode/shim/chaincode.go` 的 `handleGetStateByRange()` 中, 并被放入 `StateQueryIterator` 这个迭代器中。

当交易模拟器将交易模拟完毕, 即读写集填充完毕, 在交易返回给调用者之前, 会一次性对交易读写集中的数据进行抽取, 并放入应答信息 `ProposalResponse` 中, 即 11.3.1 节关于 `peer` 节点使用 `VersionedDB` 的过程中的第 6 步, 也即调用 `TxSimulator` 的 `GetTxSimulationResults()` 接口:

(1) `s.Done()`, 只能执行一次。`TxSimulator` 每次执行范围查询时, 不会像读取单个值一样直接就把查询的数据放入读集中, 而是只将迭代器存储在 `queryHelper` 的 `itrs` 中, 这些迭代器即代表了交易中读取了哪些范围内的数据。等到执行 `Done` 时, 才会依据这些迭代器, 一个迭代器对应一个范围读值, 将每个范围读取的所有读值填写到范围读集中。

(2) `s.rwsetBuilder.GetTxReadWriteSet()...`, 该函数主要用于排序, 然后换个包装。先使用 `sort.Strings(keys)` 对读集、写集、范围读集按照 `key` 进行排序, 然后将读写集从 `RWSetBuilder` 中换装到 `rwsetutil/rwset_proto_util.go` 中定义的 `TxRWSet` 里面。

3. 验证器

`state` 数据库是保存的实时的最新状态, `peer` 节点进行查询操作时, 也是从该账本读取的数据。所以在向 `state` 数据库提交交易数据前要使用验证器进行验证(对看上文 `peer` 节点使用 `VersionedDB` 的过程的第 8 步)。验证的时候, 无效的交易将被屏蔽掉而不会提交

到 state 账本中。因此需要一个验证器，这个验证器在 `validator/statebasedval/state_based_validator.go` 中实现为 `Validator`，其 `ValidateAndPrepareBatch(block, domvcc)` 接口负责实现验证功能，其中第 2 个参数 `domvcc` 决定了是否进行 mvcc 验证，默认为 `true`。关于 mvcc（多版本并发控制），各位可以自行学习。

存在验证就存在对比，因为验证只能通过对比完成。对比的内容为 key 对应的版本号。对比的双方：甲方是 block 中携带的读集和范围读集，乙方是验证时现从 state 数据库里查询出的与甲方对应的读值和范围读集 + 由甲方自带的有效交易的写集形成的升级数据包 `updates`（参看下文）。范围读集中虽说可能是哈希值，但是这些哈希值也是在原始读值（包含 key 和版本）的基础上形成的哈希值，在验证的时候，再从 state 数据库中读取同样范围内的读值集合，按照同样的方法生成一个新的哈希值。如果两个哈希值不同，则间接说明了当前 state 数据库中这个范围内的值已经有所变动，即比较哈希值其实也是在比较与 key 对应的版本号。

如现在验证的 block 中一个 `Envelope` 的读集中有两个读值 `<read key="K1", version="1">`、`<read key="K2", version="1">`，有一个范围读值 `<queryinfo, startkey=10, endkey=20, true, hash10-20>`，则验证的方法只有一个：验证时，依次将 K1、K2 的 version 与现从 state 数据库中现读出的 K1、K2 的版本号相比较，然后将 `hash10-20` 与现从 state 数据库中读出 `startkey-endkey` 间的读值并按照同样的归并哈希的方法生成的新哈希值 `hash10-20'` 进行比较。概括点说，就是拿之前模拟交易产生的读集中的版本号与 state 先有的相应的版本号进行比较。

验证的标准只有一个；若以上所有的比较均相同，则判定此交易有效；若任何一个比较不同，则判定交易无效。验证的目的是将有效交易的写集数据放入批量升级包中，准备更新 state 数据库。

关于理解为何如此判定交易的有效性，有几点说明：

（1）state 是存储当前账本所有最新有效交易的世界状态的数据库。这里的关键词是最新、有效、交易。

（2）所有 block 均为串行化后的数据。state 数据库中值的版本号由 `blockID+TxID` 组成，`blockID` 为序列递增，`TxID` 则为交易在 block 中一批交易中的相对位置（即交易在 `block.Data.Data` 这个数组中的下标值，亦是相对序列递增）。因此当一个新的有效交易的数据被提交至 state 数据库中进行更新时，该值的版本号与原值的版本号必然不一样，也即更新 state 数据库的值，值的版本号必然变化。

（3）模拟交易读出的数据都会放入读集中。这里假定调用者在执行 `chaincode` 的模拟交易时，读取出来的数据会被使用。这个也很正常，既然读出来了，就有被使用的可能。比如 `chaincode_example02.go` 中，即是先查出来 A、B 账户的余额数据，并在此基础上进行数据的加减。因此如果读集的数据失效，在此基础上得出的写集中的数据也必然无效。

（4）Fabric 对交易是并发并行处理的，且自模拟交易产生读写集到数据返回到 peer 节

点、送到 orderer、再返回 peer 的 gossip、再准备提交至 state 数据库，这个过程会耗费一段时间，而这段时间内，原有交易所使用的读出数据有可能已经被其他稍早点的交易改变。比如 Tx_1 查到的 A 的余额为 100，B 的余额为 100，A 把 50 元转给 B，则形成的读集是 A-v1，B-v1，写集是 A-50，B-150。这个读写集在到达 Validator 验证之前，经历了一段时间。就在这一段时间内，稍早点的交易 Tx_0 已经把 A 的 20 元转给了 B，A 的余额为 80（假定版本变为 v2），B 的余额为 120（假定版本变为 v2）。如果此时依然判定 Tx_1 有效，将 Tx_1 的写集提交至 state 数据库，则将会覆盖 Tx_0 的交易，此时 A 的余额仍然为 50（实际应该是 30），B 的余额仍然为 150（实际应该是 170），这样肯定不行。

（5）可能有人会由第 4 点推出另一个疑问：既然从 Tx_1 在交易模拟器生成读写集数据到提交前验证之间会耗费一段时间，存在稍早的交易 Tx_0 已经把 state 数据更改而使得 Tx_1 无效的风险，那么 Tx_1 在通过验证到真正提交到 state 数据库中依然存在一段时间，这期间也可能存在被稍早的交易 Tx_N 将 state 数据更改的风险。其实不存在 Tx_N 这种情况，因为如第 2 点所说，读写集是由 block 带入的，而 block 均为串行化依次处理的，这样的话只会存在两种情况：① Tx_0 与 Tx_1 不在同一个 block 中，则 Tx_0 所在的 block 早于 Tx_1 所在的 block。此时一定是 Tx_0 所在的 block 处理完毕之后，才会开始处理 Tx_1 所在的 block，也因此当处理到 Tx_1 开始验证时，此时从 state 数据库中取出的版本号必定是 Tx_0 提交过后的新的版本号 v2，也因此 Tx_1（版本号为 v1）会被判定为无效交易。② Tx_0 与 Tx_1 在同一个 block 中，则 Tx_0 所在的位置（即下标值）小于 Tx_1 所在的位置。这种情况会先验证 Tx_0 交易，如果 Tx_0 有效，则会把 Tx_0 的写集放入一个 updates 升级包中（但此时这个升级包并未提交到 state 数据库中）；接着会继续验证 Tx_1，会发现 updates 中存在与 Tx_1 读集相同的 key，也即说明在 Tx_1 之前的 Tx_0 已经在试图改变 Tx_1 所使用到的数据，而 Tx_0 又是有效且早于 Tx_1 的，因此要判定 Tx_1 无效。

4. 验证过程

验证过程还是很值得看的，因为通过验证的过程不仅可以了解数据的包装结构，还能很清楚知道什么情况下是对的、什么情况下是错的、都需要哪些对象参与等，这些对于整个业务逻辑和流程的理解都很有帮助。其实 committer 将 block 提交至账本处时已经使用自身的验证器执行了一轮验证（参看 core/committer/txvalidator/validator.go 中的 Validate(block)），并对非法的无效交易进行记录，不过该轮验证主要集中在对 block 的身份、签名、权限等内容的验证。而这里所讲的是账本中对数据的 mvcc 等验证，以确保有效交易能正确存储到 state 数据库中，所以验证过程以 validator/statebasedval/state_based_validator.go 中的 ValidateAndPrepareBatch(block, domvcc) 为起点（domvcc 默认为 true）。从函数名即可理解这个函数分为上下两部分：Validate 和 PrepareBatch，即验证和准备批量升级包 updates。

（1）updates := statedb.NewUpdateBatch()，准备 updates 用于存放批量升

级包, 这个 updates 就是上文提到的升级包, 会以参数的形式传递到每个更具体的验证函数中, 作为乙方的一部分参与验证 -> txsFilter := util.TxValidationFlags(...). 从 block 中抽取出 block 元数据的 BlockMetadataIndex_TRANSACTIONS_FILTER 位的数据 txsFilter, 该数据标记了 block 中的每笔交易有效性 (这里的有效性数据即是 committer 处验证的结果)。若是无效交易, 这里的验证将直接跳过。交易的有效值为在 protos/peer/transaction.pb.go 中定义的常量, 如 TxValidationCode_VALID 系列常量。

(2) for txIndex, envBytes := range block.Data.Data { ... }, 依次抽取 block 中的每个 Envelope, 开始验证 -> 一系列解压抽取 Envelope 的工作后, if txType != common.HeaderType_ENDORSER_TRANSACTION, 判定类型。若不是正常的交易数据, 则直接跳过, state 只存储交易数据。最重要的一步, txRWSet, txResult, err := v.validateEndorserTX(env...), 验证交易, 返回交易的读写集和验证结果 -> txsFilter.SetFlag(txIndex, txResult), 在 txsFilter 中存储当前交易的验证结果, if txRWSet != nil { ... }, 如果读写集不为空, 则说明当前交易有效, 则 committingTxHeight := version.NewHeight(...) 生成提交版本号 (写值自身是不带版本号的, 而写值的向 state 数据提交的版本号就是在这生成的), addWriteSetToBatch(...) 将有效交易的写集和对应的提交版本号放入批量升级包 updates -> for 循环结束, block.Metadata.Metadata[..._TRANSACTIONS_FILTER] = txsFilter 替换 block 元数据的交易有效性值。

(3) 第 2 步中的 txRWSet, txResult, err := v.validateEndorserTX(env) 处, 就是一个分发任务依次验证的过程。交易读写集 txRWSet.FromProtoBytes(resp.Payload.Results) 把需要验证的读写集抽取出来后, 就把任务交给了 v.validateTx(txRWSet, updates), validateTx, 然后经过分拆又把验证读集的任务交给 validateReadSet, 把验证范围读集的任务交给 validateRangeQueries。

5. 验证读集

validateReadSet 验证读集的方法就是循环调用 validateKVRead, 一一验证每一个读值。在 validateKVRead 中, 通过 if updates.Exists(ns, kvRead.Key) 查看升级包 updates 中是否有与读值相同的 key, 若存在, 则直接判定交易无效。versionedValue, err := v.db.GetState(ns, kvRead.Key) -> committedVersion = versionedValue.Version, 当下现从 state 数据库中查出读值 key 在 state 中的版本号 committedVersion -> if !version.AreSame(...), 比较两个版本号是否一致, 若不一致, 则判定当前交易无效, 直接返回 false, 否则返回 true。

6. 验证范围读集

validateRangeQueries 验证范围读集的方法就是循环调用 validateRangeQuery, 一一验证每个范围读值。范围读值的验证由于是一系列的值, 这一系列的值既要与

state 中现存的版本号比较,也要与 updates 中已存在的 key 比较,还涉及可能范围读值中的一个 key 在 updates 和 state 中同时存在(这是这个 key 就存在两个版本号了)而选哪一个版本号来和范围读值中这个 key 的版本比较的问题,所以就稍显麻烦。为了描述简洁,下文提及的 key,既代表 key 本身,也代表与 key 对应的读值。

这里涉及 validator/statebasedval/ 下的两个工具:

❑ range_query_validator.go 中的 rangeQueryValidator: 范围查询验证器, init 接口用于初始化, validate 接口用于验证。实现为两个版本,一个是用来验证原始范围读值的 rangeQueryResultsValidator,一个是用来验证范围读值的归并哈希值的 rangeQueryHashValidator。由于多数情况下范围读值中存储的是读值集合的归并哈希值,因此这里只讲后者。

❑ combined_iterator.go 中的 combinedIterator: 联合迭代器。这个联合迭代器是供 rangeQueryValidator 管理使用的。针对上述验证的麻烦之处,联合迭代器联合的就是以 updates 为数据源生成的迭代器 A (参看 statedb/statedb.go 中的 nsIterator) 和以 state 数据库为数据源生成的迭代器 B (参看 statedb/stateleveldb/stateleveldb.go 中的 kvScanner)。当前从 A 中获取的值存储在 updatesItem 中,当前从 B 中获取的值存储在 dbItem (类型为 VersionedKV, 包含 key、值、版本号)中。当执行联合迭代器的 Next() 获取一个查询的版本号时,联合迭代器会调用 compareKeys() 比较 updatesItem 和 dbItem 中的 key; 若 updatesItem 的 key 更小,则返回 updatesItem, A 前进一步; 若 dbItem 的 key 更小,则返回 dbItem, B 前进一步; 当两个 key 相等时返回 updatesItem, A 和 B 都前进一步。这里的前进一步,指的是迭代器执行一下 Next() 接口迭代到下一个值。

回到 validateRangeQuery 中:

(1) includeEndKey := !rangeQueryInfo.ItrExhausted, 记录下此范围读值是否包含与 endkey 对应的读值。因为范围读值默认是不包含与 endkey 对应的读值的,但是当这个范围读值没有被模拟器读尽时,比如 key2-key10 只读了前 3 个就返回了,此时 rangeQueryInfo.ItrExhausted 值为 false, 则说明此时实际的 endkey 是 key4 而非 key10, 且此时范围读值应该包含 key4 的读值。相反,若 ItrExhausted 值为 true 时,则说明模拟器读尽了 key2-key10 间的 8 个 key, 由于范围读值默认不包含 endkey, 所以此时范围读值中不包含与 key10 对应的读值。

(2) combinedItr, err := newCombinedIterator(...), 根据 updates、startkey、endkey、includeEndKey, 创建一个联合迭代器。

(3) if rangeQueryInfo.GetReadsMerkleHashes() != nil 成立(默认返回的范围读值中是归并后的哈希值) -> validator = &rangeQueryHashValidator{} -> validator.init(rangeQueryInfo, combinedItr), 根据范围读值 rangeQueryInfo 和生成的联合迭代器 combinedItr, 同时内建了一个上文提及的 RangeQueryResultsHelper 工具, 创建一个用于验

证范围读值的归并哈希值的验证器。

(4) `validator.validate()`, 使用验证器验证范围读值。粗略的验证过程就是使用 `RangeQueryResultsHelper` 工具重新一步步构建出一个与 `rangeQueryInfo` 范围相同的默克尔树并不断进行归并哈希的操作, 将得到的归并哈希值与 `rangeQueryInfo` 携带的哈希值进行比较。

举个例子, 当前交易 `Tx_1` 验证的一个范围读值 (`startkey` 为 2, `endkey` 为 6) 中有 `Tx_1_key2`、`Tx_1_key4`、`Tx_1_key5` 共 3 个 key 的读值 (归并形成的哈希值为 `Tx_1_hash2-6`); A 里有 `A_key2` 的一个准备更新的值; B 中在 `key2~key6` 这个范围内, `B_key2` 的值被稍早的交易 `Tx_0` 改动过 (`A_key2` 的版本一定比 `B_key2` 高, 因为 A 是 `updates` 中的, 又由于 `B_key2` 是被 `Tx_0` 改动的, 因此 `B_key2` 的版本号一定比 `Tx_1_key2` 的高), `B_key3` 为 `Tx_0` 新加入的, 其余与 `Tx_1` 相同, 因此有 `B_key2`, `B_key3`, `B_key4`, `B_key5` 共 4 个 key。两种类型的范围验证器验证过程如下 (参看 `validator/statebasedval/range_query_validator.go` 中两类验证器的 `validate()`):

(1) `rangeQueryResultsValidator` 的验证过程:

①验证 `Tx_1_key2` 时, 通过比较, 发现 A 和 B 中都存在 `key2`, 将会把 `A_key2` 的值返回, A、B 都向前前进一步并更新 `updatesItem` (变为 `nil`) 和 `dbItem` (变为 `B_key3` 对应的值)。因为 `B_key2` 肯定是之前的交易提交的旧数据, 而 `A_key2` 则是当前准备向 `state` 提交的更新数据, 所以这里返回 `updatesItem`。当比较 `Tx_1_key2` 与 `A_key2` 时, 会发现两者版本不同, 此时将判定交易 `Tx_1` 无效, 对 `Tx_1` 的验证也将结束。

②假如能继续, 下一个验证的是 `Tx_1_key4`, 此时 `updatesItem` 为 `nil`, `dbItem` 为与 `B_key3` 对应的 `VersionedKV`, 返回的是 `dbItem`, 此时比较 `Tx_1_key4` 与 `B_key3` 时发现一个是 `key3`、一个是 `key4`, 此时也将判定交易 `Tx_1` 无效。因为无论这个 `key3` 是从 `updates` 冒出来的还是从 `state` 数据库中冒出来的, 都说明 `key2~key5` 这个范围内出现了新的数据, 而 `Tx_1` 中的这个范围的读值将变得无效。因为既然这个范围读值被读出来, 就有被当作一个整体使用的可能, 比如 `key2~key5` 之间全部都是某一公司的财产, 此时要读出该公司财产总和并计算出其总和的 20% 让渡给 B 公司, 模拟交易时得到的是有 100 万, 而到验证前又有一笔 10 万的入账。若此时还依然以 100 万的 20% 进行让渡显然是不合适的。另外其实像突然冒出的 `key3` 这种情况, 比较专业的说法是数据库方面的幻读 (`phantom read`) 现象, 读者可以自行搜索更全的例子, 也可参看 `state_based_validator_test.go` 中的 `TestPhantomValidation` 函数。

(2) `rangeQueryHashValidator` 的验证过程: 设 `Tx_1` 范围读值 `maxDegree` 为 2, 则形成归并哈希后的 `MaxLevel` 为 2 (参看上文归并默克尔树归并哈希的操作)。`inMerkle` 是例子中 `Tx_1` 的一个范围读值中的归并数据 (包含 `Tx_1_hash2-6`、`maxDegree`、`MaxLevel` 等信息)。在 `for { ... }` 循环中, `result, err = itr.Next()` 使用联合迭代器获取一个读值 `result`, 此时 `result` 是 `A_key2` 的值, 之后依次会是 `B_key3`、`B_key4` -> `v.resultsHelper.AddResult(...)`, 使用 `RangeQueryResultsHelper` 工具将 `result` 依次传化为适当形式后添加到默克尔树

中 -> merkle := v.resultsHelper.GetMerkleSummary(), 获取新建的默克尔树中的归并数据 merkle -> if merkle.MaxLevel < inMerkle.MaxLevel. 当默克尔树添加 A_key2、B_key3 时, 因为没有触发归并哈希值操作, 所以 merkle.MaxLevel 依然是默认的 1, 因此会在这里直接 continue。当加入 B_key4 时, 此时叶节点已经有 3 个值, 且大于 maxDegree, 故会触发归并哈希值的操作生成一个哈希值 hash2-4, merkle.MaxLevel 变为 2, 因此程序会继续向下执行 -> if lastMatchedIndex == len(merkle.MaxLevelHashes)-1 是 -1==0 的比较, 不成立 -> lastMatchedIndex++ 后变为 0 -> !bytes.Equal(...), 此时 merkle 与 inMerkle 的 MaxLevelHashes[0] 处应该都是空的, 因此相等, 程序继续循环。result, err = itr.Next(), 此次获取的 result 是 B_key5 -> 过程如上, 添加 B_key5 并不会触发 merkle 新的归并哈希值操作, 因此之后的几个 if 分支均不会进入而返回, 程序会进入下一次循环 -> result, err = itr.Next(), 此次获取的 result 为 nil, 因此进入 if result == nil {...} 分支 -> _, merkle, err = v.resultsHelper.Done() 执行最后一次归并哈希值的操作生成归并数据, 此归并数据其实是包含了之前加入的 A_key2、B_key3、B_key4、B_key5 四个值的归并哈希值。而 Tx_1_hash2-6 是 Tx_1_key2、Tx_1_key4、Tx_1_key5 三个值的归并哈希值, 如此 equals := inMerkle.Equal(merkle) 比较的话, 自然 equals 为 false, 因此返回 equals, nil, 判定 Tx_1 交易无效。

回归到 state_based_validator.go 中的 ValidateAndPrepareBatch(...), 当 validateEndorserTX 调用的 v.validateTx(txRWSet, updates) 并返回为 peer.TxValidationCode_VALID 时, 则说明经过上述验证读集、验证范围读集的过程, 当前交易有效且可以写入 state 数据库, 因此会把交易的读写集以不为 nil 的形式返回。至此 ValidateAndPrepareBatch 的 Validate 部分的工作算是完成。接着, 在 ValidateAndPrepareBatch(...) 中会进入 if txRWSet != nil {...} 分支, 完成 PrepareBatch 部分的工作。最后, 将准备的升级包 updates 返回。该 updates 返回后会被存储到 Lock-BasedTxMgr 的 batch 成员, 供下一步 l.txtmgmt.Commit() 向 state 数据提交这些升级数据的时候使用 (参看 11.3.1 节第 8 步)。在 l.txtmgmt.Commit() 中 (txmgr/lockbasedtxmgr/lock-based_txmgr.go), txmgr.db.ApplyUpdates(...) 即是 state 数据库使用升级包数据来真正升级状态的。这里传入了两个参数: 一个是 txmgr.batch, 即升级数据包; 另一个是以 blockID+block 最后一个交易的下标序号组成的版本号 s_version。

7. state 数据库

state 数据库就是一个正常的 leveldb 数据库, 实现为 statedb/stateleveldb/stateleveldb.go 中的 versionedDB。提供基本的打开、关闭、查询、写入功能, 此外还支持范围查询的 leveldb 数据库的迭代器, 实现多值查询的额外功能。leveldb 版本的 state 数据不支持富查询, 即 ExecuteQuery(...) 接口直接返回错误。

- ❑ 查: GetState, GetStateMultipleKeys 的两个接口分别提供单值查询、多值查询。GetStateRangeScanIterator 接口提供范围查询的迭代器。

□ 写：state 数据库为 LockBasedTxMgr 配合处理 block，因此只支持批量升级数据，即 `ApplyUpdates(batch,height)` 接口的功能。batch 为上文验证过程中准备的有效交易的升级数据包，height 则为一个版本号（即上文的 `s_version`），该版本号不用于具体的某个状态，而是作为 state 数据库的一个名为保存点的键值对的值（`idStore`、`BlockStore` 中都有类似的保存点，也可以叫检查点）。具体的写入操作也是常规的 `leveldb` 数据库的批量写入操作，保存点会在每一批升级数据的最后加入，key 为 `savePointKey`，value 为 height。也就是说，当能从 state 中获取保存点的 height 值，且 height 中的 `BlockNum` 为 N 时，则说明当前 state 数据库已经完整保存了序号为 N 的 block 中的有效交易。

11.3.3 重启恢复

同 `idStore` 和 `BlockStore` 存储一样，`VersionedDB` 同样存在宕机重启后残缺数据的清理和恢复问题。不同于 `idStore` 和 `BlockStore` 的在自身对象建立起来后即进行自我恢复，`VersionedDB` 需要在 `BlockStore` 自我恢复完毕之后，由 `kvLedger` 使用恢复后的 `BlockStore` 提供的数据进行手动的恢复。为何使用 `BlockStore` 提供的数据来恢复呢？这点需要参看账本处理 block 的前后顺序：参看 `core/ledger/kvledger/kv_ledger.go` 的 `Commit(block)`，是先执行 `l.blockStore.AddBlock(block)` 把 block 提交至 `BlockStore` 中后，然后执行 `l.txnmgmt.Commit()` 将有效交易数据提交至 state 数据库，最后向 `HistoryDB` 账本提交（这个提交顺序很重要，对恢复各个账本时的逻辑有很大影响）。`BlockStore` 亦用于自我恢复，在运行 `newKVLedger(...)` 之时，传入的 `BlockStore` 对象其实是已经完成了自我修复。因此当宕机发生时，`BlockStore` 中的数据可能是比 `VersionedDB` 更新的内容。在 `newKVLedger(...)` 中，随后执行了 `l.recoverDBs()`，以此来恢复 `VersionedDB` 和 `HistoryDB`。`HistoryDB` 在下文详述。

在 `recoverDBs()` 中，`lastAvailableBlockNum` 是从 `BlockStore` 中获取的最新且有效的 `blockID`，`recoverables` 存放预计需要恢复的账本对象，`recoverers` 存放真正需要恢复的账本对象：

(1) `info, _ := l.blockStore.GetBlockchainInfo(), lastAvailableBlockNum := info.Height - 1`，使用已恢复的 `BlockStore` 获取当前已经写入 `BlockStore` 的有效的 `blockID`。

(2) `recoverables := []recoverable{...}`，可能需要恢复的有 `VersionedDB` 和 `HistoryDB` 两个数据库。

(3) `for _, recoverable := range recoverables {...}`，依次调用 `recoverable.ShouldRecover(lastAvailableBlockNum)`，使用最新有效的 `lastAvailableBlockNum`，检测账本是否需要恢复。这里只看 `VersionedDB`（`HistoryDB` 的检测方式与之相同）。在 `txmgr/lockbasedtxmgr/lockbased_txmgr.go` 的 `ShouldRecover` 中，

`savepoint, err := txmgr.GetLastSavepoint()` 获取了 `state` 数据库中的保存点；然后将 `savepoint.BlockNum` 与 `lastAvailableBlockNum` 进行对比，如果相等，则 `state` 数据库不需要恢复，如果不相等（`savepoint.BlockNum` 一定比 `lastAvailableBlockNum` 小），则说明 `state` 中未完整存储序号为 `lastAvailableBlockNum` 的 `block` 的所有有效交易，需要进行恢复；当需要恢复时，将返回 `true`，`savepoint.BlockNum + 1`。返回 `true` 表示 `state` 需要恢复操作，`savepoint.BlockNum + 1` 则表示需要从哪一块 `block` 进行恢复，即恢复起点。`ShouldRecover` 结束后，返回的数据付给了 `recoverFlag`、`firstBlockNum`，如果 `recoverFlag` 为 `true`，则会将 `VersionedDB` 账本放入 `recoverers`。这里假设 `VersionedDB` 确实需要恢复。

（4）接着，也就是 `for` 循环之后，有一系列的 `if` 分支。`if len(recoverers) == 0` 成立，说明没有确实需要恢复的账本，直接返回。`if len(recoverers) == 1` 成立，则说明只有一个需要恢复的账本，此时只可能是 `HistoryDB`，因为若 `VersionedDB` 需要恢复，则 `HistoryDB` 必也需要恢复，且 `VersionedDB` 的恢复起点一定不小于 `HistoryDB` 的恢复起点。因此这里只看两个账本都需要恢复的情况，设 `VersionedDB` 需要从 10 处开始恢复，`HistoryDB` 需要从 6 处开始恢复，`lastAvailableBlockNum` 值为 15，则：

① `if [0]... > [1]...` 将成立，因此执行 `[0], [1] = [1], [0]` 置换，将 `lagger` 放入 `recoverers` 的 0 的位置，`lagger` 的意思就是懒散的、落伍的人的意思，这里指恢复起点值更小的账本，且肯定是 `HistoryDB`，而把恢复起点值更大的 `VersionedDB` 放到后边。

② `if [0]... != [1]...`，置换后，0 处的 6 肯定不等于 1 处的 10，因此执行 `l.recommitLostBlocks(...)`，单独向 `lagger` 提交 6~9 之间的 `block` 数据进行恢复。

③ `l.recommitLostBlocks([1]..., lastAvailableBlockNum, [0], [1])`，向 `lagger` 和 `VersionedDB` 一同提交 10~15 之间的 `block` 数据进行恢复。当两个账本的恢复起点相等时，比如都为 8，则①和②处的 `if` 分支都不会进入而直接进行此步，一起向两个账本提交 8~15 之间的 `block` 数据进行恢复。这里需要说明一下，当两个账本的恢复起点不一致时为什么要进行①和②的操作，又是换位置又是分段恢复的：因为 `block` 是存储在磁盘文件 `blockfile` 中的，因此换位置和分段恢复的操作，可以使得恢复过程中 `BlockStore` 读取 `blockfile` 中的 `block` 数据时顺序读取一次，这样效率更高。

（5）最后单独看一下 `recommitLostBlocks(...)`，传入的参数分别为恢复起点 `firstBlockNum`、恢复终点 `lastBlockNum`、需要恢复的账本 `recoverables`。`for` 循环中，`block, err = l.GetBlockByNumber(blockNumber)` 使用 `BlockStore` 从 `blockfile` 中获取指定序列号的 `block`，然后调用账本的 `r.CommitLostBlock(block)`，向账本提交 `block`，进行恢复。这里只看 `VersionedDB`。在 `txmgr/lockbasedtxmgr/lockbased_txmgr.go` 的 `CommitLostBlock(block)` 中，和正常地向 `VersionedDB` 账本提交交易数据的过程如出一辙：`txmgr.ValidateAndPrepare(block, false)` 准备升级包数据。注意这里第二个参数给的是 `false`，即不再做 `mvcc` 验证，因为这里是恢复，所提交的 `block` 是从 `BlockStore` 中获取的，也肯定是之前已经验证过才会放入 `BlockStore` 的，所以这里不需要

再重复进行验证。txmgr.Commit() 将升级包数据真正提交到 state 数据库，完成 Versioned-DB 的恢复工作。

11.4 Ledger 之 HistoryDB

HistoryDB 也是一个标准的以 leveldb 数据库为依托的账本，实现在 core/ledger/kvledger/history/historydb 下（下文以此目录为基准）。比较特殊的地方是，这个账本只存储 block 中有效交易的 key，而不存储 value（由于 leveldb 存储的键值对不允许 nil，因此这里实际上 value 是有值的，只不过所有的值均为 []byte{}）。另外提供一个历史查询器 HistoryDBQueryExecutor 在交易的时候使用。其余的操作（如开闭、读写操作）均与正常的 leveldb 类型账本的对应操作类似。

HistoryDB 是可配置用来确定是否使用的，在 core/ledger/ledgerconfig/ledger_config.go 的 IsHistoryDBEnabled() 接口可以获取当前是否开启 HistoryDB。原始的配置是通过 core.yaml 中的 enableHistoryDatabase 项实现的，该项默认为 true，即开启 HistoryDB。

HistoryDB 实际只存储 key，意味着 key 自身既携带了索引信息和我们需要的值信息。key 其实就是 HistoryDB 所要存储的等价于 value 的对象并供外界检索。这个 key 是一个组合 key，检查 historyleveldb/historyleveldb.go 中的 Commit(block) 接口，当向 HistoryDB 提交一个 block 时，会筛选出 block 中的有效交易，并把这些交易的写集中的每个写值读取出来，以“命名空间 ns + compositeKeySep + 写值 key + compositeKeySep + block 序列号 + 交易 ID”的组合形式形成一个组合键 compositeHistoryKey，然后通过 dbBatch.Put(compositeHistoryKey, emptyValue)，将 compositeHistoryKey 与空值 emptyValue 作为一个键值对写入批量升级包 dbBatch 中。当 block 中所有有效交易均遍历完毕后，dbBatch.Put(savePointKey, height.ToBytes()) 以保存点封底，最后 historyDB.db.WriteBatch(dbBatch, false) 向 HistoryDB 提交数据。

组合键 compositeHistoryKey 中的 compositeKeySep 当作分隔符理解即可。前半部分“命名空间 ns + compositeKeySep + 写值 key + compositeKeySep”即为用于索引的信息，后半部分“block 序列号 + 交易 ID”即为值信息。

11.4.1 历史查询器

对 HistoryDB 的检索主要通过 HistoryDB 提供的一个 HistoryDBQueryExecutor 对象来实现的。HistoryQueryExecutor 在 historyleveldb/historyleveldb_query_executor.go 中实现，只提供 GetHistoryForKey(...) 一个接口，该接口根据提供的命名空间和写值 key，返回一个迭代器 historyScanner（内部封装了 leveldb 数据库的 Iterator）。迭代器必定涉及起点和止点，historyScanner 的起点是“命名空间 ns + compositeKeySep + 写值 key + compositeKeySep”的组合键，止点是“命名空间 ns + compositeKeySep + 写值 key +

compositeKeySep + 0xff”的组合键。对比起点和止点，止点多了一个 0xff，相当于一个字符的极限值，也因此这个范围查询的是所有以“命名空间 ns + compositeKeySep + 写值 key + compositeKeySep”为开头的 key 值。比如起点是 []byte("A")，止点是 append([]byte("A"), []byte{0xff}...)，则这个范围查询的是所有以字符 A 开头的 key。又因为 HistoryDB 存储的格式为前提，假设这里的 ns 为“chaincode_example02”，写值 key 为“A 账户”，则这个起止范围相当于在查询所有 chaincode_example02 链码上改动过 A 账户的“blockID+有效交易 ID”的信息。而有了 blockID+有效交易 ID 这两个信息，自然可以通过 BlockStore 定位查询出原交易的所有信息，如改动时间、改动值、是否是删除操作等。

参看 historyleveldb_query_executer.go 中 historyScanner 的 Next() 接口：

(1) if !scanner.dbItr.Next(), 因为 HistoryDB 不存储 value，因此这里 leveldb 的迭代器 dbItr 的 Next() 操作不为获取 value，而只是让迭代器向前走一步，同时进行是否还有下一个值的判断。

(2) historyKey := scanner.dbItr.Key(), 获取 leveldb 迭代器当前值的 key，这也是我们想获取的内容，这个 key 就是由“命名空间 ns + compositeKeySep + 写值 key + compositeKeySep + block 序列号 + 交易 ID”组成的组合键。

(3) blockNum, bytesConsumed := util..., tranNum, _ := util..., 从组合键中分解出其携带的 blockID 和交易 ID。

(4) tranEnvelope, err := scanner.blockStore..., 使用 BlockStore，根据 blockID 和交易 ID，获取原始的交易信息。

(5) queryResult, err := getKeyModificationFromTran(...), 根据获取的原始交易信息进行整理，返回当次 Next() 的单个查询结果，该结果里面包含改动时间、改动值、是否是删除操作等信息。

11.4.2 使用

HistoryDB 和 VersionedDB 被 kvLedger 使用的方式在过程上就是一先一后的区别，不用赘述。被 chaincode 使用主要是通过出现在交易中的 HistoryDBQueryExecutor 进行查询，而查询的过程与 VersionedDB 的交易模拟器的范围查询过程颇为类似（至于 HistoryDBQueryExecutor 为何会出现在交易中，请参看 core/endorser/endorser.go 的 ProcessProposal(...) 中的 ctx = context.WithValue(...) 处），以 map.go 为例：

(1) peer chaincode invoke..., 在 peer 节点执行 map 的 Invoke 命令，中间过程省略，直接定位到下一步。

(2) 在 map.go 的 Invoke(stub) 的 case "history": 分支中，通过 keysIter, err := stub.GetHistoryForKey(key) 调用 ChaincodeStubInterface 的接口，最终获取一个迭代器，该迭代器以 HistoryDBQueryExecutor 获取的范围数据为数据源并返回查询结果。

(3) 在第1步调用后, 会在 ShimHandler 和 ServerHandler 间辗转, 然后在 core/chaincode/handler.go 中调用 handleGetHistoryForKey(...) 函数中, historyIter, err := txContext.historyQueryExecutor.GetHistoryForKey(chaincodeID, getHistoryForKey.Key) 使用 HistoryDBQueryExecutor 获取一个 historyScanner, 然后在 getQueryResponse(...) 中依次调用 historyScanner 的 Next() 获取查询结果(即 11.4.1 节第 5 步查询到的内容)。查询结果最终放入了 ChaincodeMessage 的 Payload 返回。

(4) 携带查询结果的 ChaincodeMessage 返回至 core/chaincode/shim/chaincode.go 中的 GetHistoryForKey(...) 中, 也即第 1 步所调用处, 然后把结果集放在 CommonIterator 迭代器中, 上面再套上 HistoryQueryIterator 返回给 chaincode, 也就是第 1 步获取的 keysIter。

(5) 获取 keysIter 后, 就可以在 for keysIter.HasNext() { response, iterErr := keysIter.Next()... } 中进行使用了, 以完成 chaincode 在该功能上自身想完成的任务。

因此, 从使用上讲, HistoryDB 算是一个辅助性的数据库, 辅助其他数据库、辅助交易的进行。另外, HistoryDB 的组合键的组合形式可以针对自身业务的需求进行设计, 这样也可实现类似 couchDB 数据库那样的富查询。

chaincode 智能合约案例分析

本章将通过具体的案例，加深读者对 chaincode 智能合约的理解，让读者了解到 Fabric 中智能合约的编写方法。

12.1 encc_example

EncCC 是一个简单的加密 / 签名的 chaincode 实现，实现了在智能合约层面进行 AES 对称加密以及 ECDSA 签名验签的功能。

智能合约主流程在 enccc_example.go 中，相关功能函数的封装则是在 utils.go 中实现。

12.1.1 chaincode 代码分析

首先，来看一段简短的代码：

```
//chaincode结构体
type EncCC struct {
    bccspInst bccsp.BCCSP
}
// 初始化
func (t *EncCC) Init(stub shim.ChaincodeStubInterface) pb.Response {
    .....
}
//加密方法
func (t *EncCC) Encrypter(stub shim.ChaincodeStubInterface, args []string,
    encKey, IV []byte) pb.Response {
    .....
}
```



```

}
//解密方法
func (t *EncCC) Decrypter(stub shim.ChaincodeStubInterface, args []string,
    decKey, IV []byte) pb.Response {
    .....
}
//签名并加密
func (t *EncCC) EncrypterSigner(stub shim.ChaincodeStubInterface, args []string,
    encKey, sigKey []byte) pb.Response {
    .....
}
//解密并验签
func (t *EncCC) DecrypterVerify(stub shim.ChaincodeStubInterface, args []string,
    decKey, verKey []byte) pb.Response {
    .....
}
//基于范围进行解密
func (t *EncCC) RangeDecrypter(stub shim.ChaincodeStubInterface, decKey []byte)
    pb.Response {
    .....
}
//chaincode调用的主入口
func (t *EncCC) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    switch f {
    case "ENCRYPT":
    case "DECRYPT":

    case "ENCRYPTSIGN":
    case "DECRYPTVERIFY":
    case "RANGEQUERY":
    default:
        return shim.Error(fmt.Sprintf("Unsupported function %s", f))
    }
}
func main() {
    .....
}

```

上述代码为简化后的代码，便于阅读。

上述代码中包括 chaincode 的结构体，其包含了 BCCSP 接口，以便于使用 Fabric 既有的加密签名功能、相关加密解密、签名验签方法及链码调用的主入口 Invoke 方法。

(1) Invoke 方法：根据传入的 function 的值的不同，分别执行 ENCRYPT、DECRYPT、ENCRYPTSIGN、DECRYPTVERIFY、RANGEQUERY 这 5 个不同的方法。

(2) Encrypter 方法：

```

func (t *EncCC) Encrypter(stub shim.ChaincodeStubInterface, args []string,
    encKey, IV []byte) pb.Response {
    ent, err := entities.NewAES256EncrypterEntity("ID", t.bccspInst, encKey, IV)
    if err != nil {

```

```

return shim.Error(fmt.Sprintf("entities.NewAES256EncrypterEntity failed,
    err %s", err))
}
if len(args) != 2 {
return shim.Error("Expected 2 parameters to function Encrypter")
}
key := args[0]
cleartextValue := []byte(args[1])
err = encryptAndPutState(stub, ent, key, cleartextValue)
if err != nil {
return shim.Error(fmt.Sprintf("encryptAndPutState failed, err %+v", err))
}
return shim.Success(nil)
}
...

```

该方法主要实例化一个 'BCCSPEncrypterEntity', 调用 utils.go 中的 'encryptAndPutState' 函数对传入的键值对进行加密进行写操作 (需要等 committer 节点校验通过才会更新账本)。

Decrypter 方法

```

```go
func (t *EncCC) Decrypter(stub shim.ChaincodeStubInterface, args []string,
 decKey, IV []byte) pb.Response {
 ent, err := entities.NewAES256EncrypterEntity("ID", t.bccspInst, decKey, IV)
 if err != nil {
 return shim.Error(fmt.Sprintf("entities.NewAES256EncrypterEntity failed,
 err %s", err))
 }
 if len(args) != 1 {
 return shim.Error("Expected 1 parameters to function Decrypter")
 }
 key := args[0]
 cleartextValue, err := getStateAndDecrypt(stub, ent, key)
 if err != nil {
 return shim.Error(fmt.Sprintf("getStateAndDecrypt failed, err %+v", err))
 }
 return shim.Success(cleartextValue)
}

```

该方法主要实例化一个 BCCSPEncrypterEntity, 调用 utils.go 中的 getStateAndDecrypt 函数, 进行查询并解密返回。

### (3) EncrypterSigner 方法:

```

func (t *EncCC) EncrypterSigner(stub shim.ChaincodeStubInterface, args []string,
 encKey, sigKey []byte) pb.Response {
 ent, err := entities.NewAES256EncrypterECDSASignerEntity("ID", t.bccspInst,
 encKey, sigKey)
 if err != nil {

```

```

 return shim.Error(fmt.Sprintf("entities.NewAES256EncrypterEntity failed,
 err %s", err))
 }
 if len(args) != 2 {
 return shim.Error("Expected 2 parameters to function EncrypterSigner")
 }
 key := args[0]
 cleartextValue := []byte(args[1])
 err = signEncryptAndPutState(stub, ent, key, cleartextValue)
 if err != nil {
 return shim.Error(fmt.Sprintf("signEncryptAndPutState failed, err %+v",
err))
 }
 return shim.Success(nil)
}

```

该方法调用 `utils.go` 中的 `signEncryptAndPutState` 函数进行消息的签名以及加密。

#### (4) DecrypterVerify 方法:

```

func (t *EncCC) DecrypterVerify(stub shim.ChaincodeStubInterface, args []string,
 decKey, verKey []byte) pb.Response {
 ent, err := entities.NewAES256EncrypterECDSASignerEntity("ID", t.bccspInst,
 decKey, verKey)
 if err != nil {
 return shim.Error(fmt.Sprintf("entities.NewAES256DecrypterEntity failed,
 err %s", err))
 }
 if len(args) != 1 {
 return shim.Error("Expected 1 parameters to function DecrypterVerify")
 }
 key := args[0]
 cleartextValue, err := getStateDecryptAndVerify(stub, ent, key)
 if err != nil {
 return shim.Error(fmt.Sprintf("getStateDecryptAndVerify failed, err %+v", err))
 }
 return shim.Success(cleartextValue)
}

```

该方法调用 `utils.go` 中的 `getStateDecryptAndVerify` 函数进行消息的解密并验签。

#### (5) RangeDecrypter 方法:

```

func (t *EncCC) RangeDecrypter(stub shim.ChaincodeStubInterface, decKey []byte)
 pb.Response {
 ent, err := entities.NewAES256EncrypterEntity("ID", t.bccspInst, decKey, nil)
 if err != nil {
 return shim.Error(fmt.Sprintf("entities.NewAES256EncrypterEntity failed,
 err %s", err))
 }
 bytes, err := getStateByRangeAndDecrypt(stub, ent, "", "")
 if err != nil {

```



```

 return shim.Error(fmt.Sprintf("getStateByRangeAndDecrypt failed, err
 %+v", err))
 }
 return shim.Success(bytes)
}

```

该方法调用 `utils.go` 中的 `getStateByRangeAndDecrypt` 函数进行消息的批量解密。

### 12.1.2 使用 EncCC

为了测试 `EncCC`，你首先需要生成 AES 的 256 位的基于 `base64` 编码的字符串，这样密钥才能基于 `peer chaincode invoke` 的 `transient` 传递参数。

在开始之前，你必须使用 `govendor` 来添加额外的依赖。应在 `enccc_example` 目录执行下述命令：

```

govendor init
govendor add +external


```

然后生成加密解密的密钥。这个例子会模拟一个用于加解密的共享密钥。

```
ENCKEY=`openssl rand 32 -base64` && DECKEY=$ENCKEY
```

此时，你可以按照以下命令去调用 `chaincode` 对键值对进行加密操作：

---

 **注意** 下面的操作是基于环境已经初始化完成并且 `peer` 节点已经加入 `my-ch channel` 的前提下。

---

```

peer chaincode invoke -n enccc -C my-ch -c '{"Args":["ENCRYPT","key1","value1"]}'
--transient "{\"ENCKEY\":\"$ENCKEY\"}"

```

这次操作会使用一个随机的初始向量进行加密。这可能是一个不受欢迎的操作。举个例子，如果链码的调用需要被多个 `peer` 节点背书，这将导致背书的读写集冲突。如果使用一个确切的初始向量进行加密，一个 `IV` 必须被事先创建。

```
IV=`openssl rand 16 -base64`
```

然后，`IV` 必须在 `transient` 字段被明确。

```

peer chaincode invoke -n enccc -C my-ch -c '{"Args":["ENCRYPT","key2","value2"]}'
--transient "{\"ENCKEY\":\"$ENCKEY\",\"IV\":\"$IV\"}"

```

2 次这样的调用会产生相同的被多个节点背书的 KVS 写操作。

这些值会被下述操作恢复：

```

peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPT","key1"]}'
--transient "{\"DECKEY\":\"$DECKEY\"}"
peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPT","key2"]}'
--transient "{\"DECKEY\":\"$DECKEY\"}"

```



在这种情况下我们使用 `chaincode` 查询操作；使用 `transient` 字段保证了内容不会被写到账本上，`chaincode` 对消息进行解密并且放到提案的响应中。一次调用对所有的通道中的读者来说都会被持久化到账本中，而查询则会丢弃，因此结果依旧是保密的。

为了测试签名和验签，你需要为椭圆曲线生成一个 ECDSA 密钥，具体如下：

On Intel:

```
SIGKEY=`openssl ecparam -name prime256v1 -genkey | tail -n5 | base64 -w0` &&
VERKEY=$SIGKEY
```

On Mac:

```
SIGKEY=`openssl ecparam -name prime256v1 -genkey | tail -n5 | base64` &&
VERKEY=$SIGKEY
```

此时，你可以调用 `chaincode` 去签名加密键值对，具体如下：

```
peer chaincode invoke -n enccc -C my-ch -c '{"Args":["ENCRYPTSIGN","key3","value3"]}' --transient "{\"ENCKEY\":\"$ENCKEY\",\"SIGKEY\":\"$SIGKEY\"}"
```

相似的，你可以使用查询操作去检索它们。

```
peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPTVERIFY","key3"]}'
--transient "{\"DECKEY\":\"$DECKEY\",\"VERKEY\":\"$VERKEY\"}"
```

`enccc_example` 这个 `chaincode` 展示了在 `fabric` 中基于智能合约进行数据加解密及签名验签的过程，关于加解密及签名验签相关逻辑可参考 [github.com/hyperledger/fabric](https://github.com/hyperledger/fabric) 及 [github.com/hyperledger/fabric](https://github.com/hyperledger/fabric) 的 `bccsp` 及 `core/chaincode/shim/ext/entities` 两个包中的相关实现。

## 12.2 eventsender

`eventsender` 是一个用于展示 `Fabric` 事件订阅的智能合约。当被调用的这个交易被 `commmitter` 验证通过写入区块时会触发该智能合约中所发送的事件。

`chaincode` 代码分析如下：

```
type EventSender struct {
}
//初始化方法
func (t *EventSender) Init(stub shim.ChaincodeStubInterface) pb.Response {
 err := stub.PutState("noevents", []byte("0"))
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(nil)
}
```

```

// 调用方法
func (t *EventSender) invoke(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
 b, err := stub.GetState("noevents")
 if err != nil {
 return shim.Error("Failed to get state")
 }
 noevts, _ := strconv.Atoi(string(b))

 tosend := "Event " + string(b)
 for _, s := range args {
 tosend = tosend + "," + s
 }

 err = stub.PutState("noevents", []byte(strconv.Itoa(noevts+1)))
 if err != nil {
 return shim.Error(err.Error())
 }

 err = stub.SetEvent("evtsender", []byte(tosend))
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(nil)
}

// 查询方法
func (t *EventSender) query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
 b, err := stub.GetState("noevents")
 if err != nil {
 return shim.Error("Failed to get state")
 }
 jsonResp := "{\"NoEvents\":\"" + string(b) + "\"}"
 return shim.Success([]byte(jsonResp))
}

func (t *EventSender) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 if function == "invoke" {
 return t.invoke(stub, args)
 } else if function == "query" {
 return t.query(stub, args)
 }
 return shim.Error("Invalid invoke function name. Expecting \"invoke\"
 \"query\"")
}

func main() {

}

```

EventSender 链码将每次调用所产生的递增的数字以及传入的参数作为 event 的内容。



evtsender 调用 stub.PutState，将会在 committer 节点写入区块的时候触发 event 事件。可以使用 SDK 中封装的方法监听所发出的 event 事件，也可以使用 examples/events/block-listener 进行监听，这些会在后续章节进行讲解。

## 12.3 example01

在 example01 中，chaincode 描述了在初始化方法 Init 中初始化 A、B 的余额、调用 invoke 方法对 A、B 的值进行修改。该链码没有操作账本，所以这个例子并不需要更新账本。

chaincode 代码分析如下：

```
type SimpleChaincode struct{}
var A, B string
var Aval, Bval, X int
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
 var err error
 _, args := stub.GetFunctionAndParameters()
 if len(args) != 4 {
 return shim.Error("Incorrect number of arguments. Expecting 4")
 }
 // 初始化链码
 A = args[0]
 Aval, err = strconv.Atoi(args[1])
 if err != nil {
 return shim.Error("Expecting integer value for asset holding")
 }
 B = args[2]
 Bval, err = strconv.Atoi(args[3])
 if err != nil {
 return shim.Error("Expecting integer value for asset holding")
 }
 fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

 return shim.Success(nil)
}
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
 X, err := strconv.Atoi(args[0])
 if err != nil {
 fmt.Printf("Error convert %s to integer: %s", args[0], err)
 return shim.Error(fmt.Sprintf("Error convert %s to integer: %s", args[0], err))
 }
 Aval = Aval - X
 Bval = Bval + X
 ts, err2 := stub.GetTxTimestamp()
 if err2 != nil {
 fmt.Printf("Error getting transaction timestamp: %s", err2)
 return shim.Error(fmt.Sprintf("Error getting transaction timestamp: %s", err2))
 }
}
```

```

 }
 fmt.Printf("Transaction Time: %v,Aval = %d, Bval = %d\n", ts, Aval, Bval)
 return shim.Success(nil)
}
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 if function == "invoke" {
 return t.invoke(stub, args)
 }
 return shim.Error("Invalid invoke function name. Expecting \"invoke\"")
}

```

## 12.4 example02

在 example02 中, chaincode 是 example01 的升级, 展示的场景还是 A、B 之间转账, 但是进行了账本状态的修改, 使得修改后的数据可以在链上持久化。

chaincode 代码分析如下:

```

type SimpleChaincode struct {
}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 if function == "invoke" {
 // 从A到B支付X单位
 return t.invoke(stub, args)
 } else if function == "delete" {
 // 从其状态中删除一个对象
 return t.delete(stub, args)
 } else if function == "query" {
 // 旧的“查询”现在在调用中实现
 return t.query(stub, args)
 }
 return shim.Error("Invalid invoke function name. Expecting \"invoke\" \"delete\" \"query\"")
}

// 从A支付到B交易X单位
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
 pb.Response {

 }

// 从其状态中删除一个对象

```

```
func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string)
pb.Response {

}

// 表示链表查询的查询回调
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {

}
```

该 chaincode 在 Init 中初始化 A、B 账户的余额，并进行 PutState 操作，在 Invoke 方法中根据 function 的类型进行转账、删除、查询操作。function 主要有 invoke、delete、query 三个方法。

### 1. invoke 方法

invoke 方法相关代码如下：

```
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
 var A, B string // 实体对象
 var Aval, Bval int // 资产持有量
 var X int // 交易价值
 var err error
 if len(args) != 3 {
 return shim.Error("Incorrect number of arguments. Expecting 3")
 }
 A = args[0]
 B = args[1]
 Avalbytes, err := stub.GetState(A)
 if err != nil {
 return shim.Error("Failed to get state")
 }
 if Avalbytes == nil {
 return shim.Error("Entity not found")
 }
 Aval, _ = strconv.Atoi(string(Avalbytes))
 Bvalbytes, err := stub.GetState(B)
 if err != nil {
 return shim.Error("Failed to get state")
 }
 if Bvalbytes == nil {
 return shim.Error("Entity not found")
 }
 Bval, _ = strconv.Atoi(string(Bvalbytes))
 X, err = strconv.Atoi(args[2])
 if err != nil {
 return shim.Error("Invalid transaction amount, expecting a integer value")
 }
 Aval = Aval - X
```



```

 Bval = Bval + X
 fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)
 err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
 if err != nil {
 return shim.Error(err.Error())
 }
 err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(nil)
}

```

该方法主要是从参数中获取转账金额，利用 GetState 读取 A、B 的历史余额，进行 A 到 B 的转账操作、状态的修改，然后通过 PutState 对 A、B 的状态进行写操作。

## 2. delete 方法

delete 方法的相关代码如下：

```

if len(args) != 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
}
A := args[0]
err := stub.DelState(A)
if err != nil {
 return shim.Error("Failed to delete state")
}
return shim.Success(nil)

```

该方法主要是进行一个状态的删除操作，利用 DelState 进行状态的删除。

## 3. query 方法

query 方法的相关代码如下：

```

func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
 pb.Response {
 var A string // Entities
 var err error
 if len(args) != 1 {
 return shim.Error("Incorrect number of arguments. Expecting name of the
 person to query")
 }
 A = args[0]
 Avalbytes, err := stub.GetState(A)
 if err != nil {
 jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
 return shim.Error(jsonResp)
 }
 if Avalbytes == nil {
 jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"

```

```

 return shim.Error(jsonResp)
 }
 jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
 fmt.Printf("Query Response:%s\n", jsonResp)
 return shim.Success(Avalbytes)
}

```

该方法主要利用 GetState 进行状态的读取。

## 12.5 example03

在 example03 中, chaincode 是一个简单的用于存值的智能合约, 将通过调用智能合约来更新合约中的状态。

chaincode 代码分析如下:

```

type SimpleChaincode struct{}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
 var A string // Entity
 var Aval int // Asset holding
 var err error
 _, args := stub.GetFunctionAndParameters()
 if len(args) != 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
 }
 A = args[0]
 Aval, err = strconv.Atoi(args[1])
 if err != nil {
 return shim.Error("Expecting integer value for asset holding")
 }
 fmt.Printf("Aval = %d\n", Aval)
 err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(nil)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 if function == "query" {
 return t.query(stub, args)
 }
 return shim.Error("Invalid invoke function name. Expecting \"query\"")
}

func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
 var A string // Entity
 var Aval int // Asset holding
 var err error

```

```

if len(args) != 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
}
A = args[0]
Aval, err = strconv.Atoi(args[1])
if err != nil {
 return shim.Error("Expecting integer value for asset holding")
}
fmt.Printf("Aval = %d\n", Aval)
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil {
 jsonResp := "{\"Error\":\"Cannot put state within chaincode query\"}"
 return shim.Error(jsonResp)
}
return shim.Success(nil)
}

```

该 chaincode 在初始化的时候进行了 A 变量状态的设定，在 query 的时候不必重新 PutState 更新 A 变量的值。

## 12.6 example04

example04 是一个跨 chaincode 调用 example02 的案例，用于展示 Fabric 中跨智能合约调用的场景。

chaincode 代码分析如下：

```

type SimpleChaincode struct{}
func toChaincodeArgs(args ...string) [][]byte {

}
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {

}
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
 pb.Response {

 }
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
 pb.Response {

 }
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {

}

```

该 chaincode 主要针对 example02 进行 A、B 转账时进行跨 chaincode 的调用以及跨 chaincode 的查询。



## 1. invoke 方法

invoke 方法的相关代码如下：

```
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
 pb.Response {
 var event string // Event entity
 var eventVal int // State of event
 var err error
 if len(args) != 3 && len(args) != 4 {
 return shim.Error("Incorrect number of arguments. Expecting 3 or 4")
 }
 chainCodeToCall := args[0]
 event = args[1]
 eventVal, err = strconv.Atoi(args[2])
 if err != nil {
 return shim.Error("Expected integer value for event state change")
 }
 channelID := ""
 if len(args) == 4 {
 channelID = args[3]
 }
 if eventVal != 1 {
 fmt.Printf("Unexpected event. Doing nothing\n")
 return shim.Success(nil)
 }
 f := "invoke"
 invokeArgs := toChaincodeArgs(f, "a", "b", "10")
 response := stub.InvokeChaincode(chainCodeToCall, invokeArgs, channelID)
 if response.Status != shim.OK {
 errStr := fmt.Sprintf("Failed to invoke chaincode. Got error: %s", string(
 response.Payload))
 fmt.Printf(errStr)
 return shim.Error(errStr)
 }
 fmt.Printf("Invoke chaincode successful. Got response %s", string(response.Payload))
 err = stub.PutState(event, []byte(strconv.Itoa(eventVal)))
 if err != nil {
 return shim.Error(err.Error())
 }
 return response
 }
```

该方法是基于 shim 层暴露出来的 stub.InvokeChaincode 进行一个跨 chaincode 的调用，操作 example02 的 chaincode，效果等价于直接调用 example02 的 invoke 方法，同样能够修改 example02 下的 A、B 账户的数据。

## 2. query 方法

query 方法的相关代码如下：

```
func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
```

```

pb.Response {
var event string // Event entity
var err error
if len(args) < 1 {
 return shim.Error("Incorrect number of arguments. Expecting entity to query")
}
event = args[0]
var jsonResp string
eventValbytes, err := stub.GetState(event)
if err != nil {
 jsonResp = "{\"Error\":\"Failed to get state for " + event + "\"}"
 return shim.Error(jsonResp)
}
if eventValbytes == nil {
 jsonResp = "{\"Error\":\"Nil value for " + event + "\"}"
 return shim.Error(jsonResp)
}
if len(args) > 3 {
 chainCodeToCall := args[1]
 queryKey := args[2]
 channel := args[3]
 f := "query"
 invokeArgs := toChaincodeArgs(f, queryKey)
 response := stub.InvokeChaincode(chainCodeToCall, invokeArgs, channel)
 if response.Status != shim.OK {
 errStr := fmt.Sprintf("Failed to invoke chaincode. Got error: %s", err.
 Error())
 fmt.Printf(errStr)
 return shim.Error(errStr)
 }
 jsonResp = string(response.Payload)
} else {
 jsonResp = "{\"Name\":\"" + event + "\", \"Amount\":\"" + string(event-
 Valbytes) + "\"}"
}
fmt.Printf("Query Response: %s\n", jsonResp)
return shim.Success([]byte(jsonResp))
}

```

该方法与 invoke 方法类似，区别在于调用 stub.InvokeChaincode 方法时调用的是 example02 的 query 方法，从 example02 中进行状态的查询。

## 12.7 example05

example05 同样也是一个跨 chaincode 调用 example02 的案例，和 example04 的区别在于它将查询获取到的 A、B 的值记录了下来。

chaincode 代码分析如下：

```

func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)

```

```

pb.Response {
var sum, channelName string // Sum entity
var Aval, Bval, sumVal int // value of sum entity - to be computed
var err error
if len(args) < 2 {
 return shim.Error("Incorrect number of arguments. Expecting atleast 2")
}
chaincodeName := args[0]
sum = args[1]
if len(args) > 2 {
 channelName = args[2]
} else {
 channelName = ""
}
f := "query"
queryArgs := toChaincodeArgs(f, "a")
response := stub.InvokeChaincode(chaincodeName, queryArgs, channelName)
if response.Status != shim.OK {
 errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s", res-
 ponse.Payload)
 fmt.Printf(errStr)
 return shim.Error(errStr)
}
Aval, err = strconv.Atoi(string(response.Payload))
if err != nil {
 errStr := fmt.Sprintf("Error retrieving state from ledger for queried chain-
 code: %s", err.Error())
 fmt.Printf(errStr)
 return shim.Error(errStr)
}
queryArgs = toChaincodeArgs(f, "b")
response = stub.InvokeChaincode(chaincodeName, queryArgs, channelName)
if response.Status != shim.OK {
 errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s", res-
 ponse.Payload)
 fmt.Printf(errStr)
 return shim.Error(errStr)
}
Bval, err = strconv.Atoi(string(response.Payload))
if err != nil {
 errStr := fmt.Sprintf("Error retrieving state from ledger for queried chain-
 code: %s", err.Error())
 fmt.Printf(errStr)
 return shim.Error(errStr)
}
sumVal = Aval + Bval
err = stub.PutState(sum, []byte(strconv.Itoa(sumVal)))
if err != nil {
 return shim.Error(err.Error())
}
fmt.Printf("Invoke chaincode successful. Got sum %d\n", sumVal)

```



```

 return shim.Success([]byte(strconv.Itoa(sumVal)))
}

```

在 `invoke` 方法中进行了 2 次跨 `chaincode` 的操作，将从 `example02` 中获取到的 A、B 的值求和存入了当前 `chaincode` 中（需要注意，跨链码调用如果要进行跨信道，则需要指明 `channelName`）。

## 12.8 invokereturnsvalue

`invokereturnsvalue` 案例主要用于展示如何对智能合约调用的返回值进行设定。

`chaincode` 代码分析如下：

```

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 _, args := stub.GetFunctionAndParameters()
 var A, B string // Entities
 var Aval, Bval int // Asset holdings
 var X int // Transaction value
 var err error
 if len(args) != 3 {
 return shim.Error("Incorrect number of arguments. Expecting 3")
 }
 A = args[0]
 B = args[1]
 Avalbytes, err := stub.GetState(A)
 if err != nil {
 return shim.Error("Failed to get state")
 }
 if Avalbytes == nil {
 return shim.Error("Entity not found")
 }
 Aval, _ = strconv.Atoi(string(Avalbytes))

 Bvalbytes, err := stub.GetState(B)
 if err != nil {
 return shim.Error("Failed to get state")
 }
 if Bvalbytes == nil {
 return shim.Error("Entity not found")
 }
 Bval, _ = strconv.Atoi(string(Bvalbytes))
 X, err = strconv.Atoi(args[2])
 if err != nil {
 return shim.Error("Invalid transaction amount, expecting a integer value")
 }
 Aval = Aval - X
 Bval = Bval + X
 err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
 if err != nil {

```

```

 return shim.Error(err.Error())
 }
 err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success([]byte(fmt.Sprintf("%d,%d", Aval, Bval)))
}

```

链码调用的返回值可以被封装在 `shim.Success()` 方法中, 该方法传入的 `[]byte` 参数即为客户端收到的 `response` 中的 `payload`。

## 12.9 map

`map` 这个案例展示了关于组合键的存取、范围查询以及调用 `couchdb` 进行富查询的操作。

`chaincode` 代码分析如下:

```

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 switch function {

 case "put":

 case "remove":

 case "get":

 case "keys":

 case "query":

 case "history":

 default:
 return shim.Success([]byte("Unsupported operation"))
 }
}

```

该链码主要分为 6 个功能。

### 1. put 方法

该方法除了调用了之前介绍的 `PutState` 进行数据存取外, 还调用了创建组合键的函数 `CreateCompositeKey` 用于 `Key` 值的生成。

```

if len(args) < 2 {
 return shim.Error("put operation must include two arguments: [key, value]")
}

```

```

 }
 key := args[0]
 value := args[1]
 if err := stub.PutState(key, []byte(value)); err != nil {
 fmt.Printf("Error putting state %s", err)
 return shim.Error(fmt.Sprintf("put operation failed. Error updating
 state: %s", err))
 }
 indexName := "compositeKeyTest"
 compositeKeyTestIndex, err := stub.CreateCompositeKey(indexName, []
 string{key})
 if err != nil {
 return shim.Error(err.Error())
 }
 valueByte := []byte{0x00}
 if err := stub.PutState(compositeKeyTestIndex, valueByte); err != nil {
 fmt.Printf("Error putting state with compositeKey %s", err)
 return shim.Error(fmt.Sprintf("put operation failed. Error updating
 state with compositeKey: %s", err))
 }
 return shim.Success(nil)
}

```

## 2. remove 方法

该方法调用 DelState 方法进行键值的删除操作。

```

if len(args) < 1 {
 return shim.Error("remove operation must include one argument: [key]")
}
key := args[0]

err := stub.DelState(key)
if err != nil {
 return shim.Error(fmt.Sprintf("remove operation failed. Error
 updating state: %s", err))
}
return shim.Success(nil)
}

```

## 3. get 方法

该方法调用 GetState 进行查询操作。

```

if len(args) < 1 {
 return shim.Error("get operation must include one argument, a key")
}
key := args[0]
value, err := stub.GetState(key)
if err != nil {
 return shim.Error(fmt.Sprintf("get operation failed. Error accessing
 state: %s", err))
}
jsonVal, err := json.Marshal(string(value))
return shim.Success(jsonVal)
}

```



## 4. keys 方法

keys 方法用于进行范围查询 GetStateByRange，根据传入的 startKey、endKey 进行字典序排列。

```

if len(args) < 2 {
 return shim.Error("put operation must include two arguments, a key
 and value")
}
startKey := args[0]
endKey := args[1]
stime := 0
if len(args) > 2 {
 stime, _ = strconv.Atoi(args[2])
}
keysIter, err := stub.GetStateByRange(startKey, endKey)
if err != nil {
 return shim.Error(fmt.Sprintf("keys operation failed. Error accessing
 state: %s", err))
}
defer keysIter.Close()
var keys []string
for keysIter.HasNext() {
 if stime > 0 {
 time.Sleep(time.Duration(stime) * time.Millisecond)
 }
 response, iterErr := keysIter.Next()
 if iterErr != nil {
 return shim.Error(fmt.Sprintf("keys operation failed. Error
 accessing state: %s", err))
 }
 keys = append(keys, response.Key)
}
for key, value := range keys {
 fmt.Printf("key %d contains %s\n", key, value)
}
jsonKeys, err := json.Marshal(keys)
if err != nil {
 return shim.Error(fmt.Sprintf("keys operation failed. Error mar-
 shaling JSON: %s", err))
}
return shim.Success(jsonKeys)

```

## 5. query 方法

该方法用 GetQueryResult 进行一个富查询操作，该接口只针对提供富查询操作的 couchdb 等 statedb，不支持 leveldb。

```

query := args[0]
keysIter, err := stub.GetQueryResult(query)
if err != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error accessing

```

```

 state: %s", err))
 }
 defer keysIter.Close()

 var keys []string
 for keysIter.HasNext() {
 response, iterErr := keysIter.Next()
 if iterErr != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error
 accessing state: %s", err))
 }
 keys = append(keys, response.Key)
 }

 jsonKeys, err := json.Marshal(keys)
 if err != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
 JSON: %s", err))
 }

 return shim.Success(jsonKeys)
}

```

## 6. history 方法

该方法用于获取键值对的历史状态变化，通过调用 `GetHistoryForKey` 方法获取 key 值每一次变化所对应的 txID。

```

key := args[0]
keysIter, err := stub.GetHistoryForKey(key)
if err != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error accessing
 state: %s", err))
}
defer keysIter.Close()
var keys []string
for keysIter.HasNext() {
 response, iterErr := keysIter.Next()
 if iterErr != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error
 accessing state: %s", err))
 }
 keys = append(keys, response.TxId)
}
for key, txID := range keys {
 fmt.Printf("key %d contains %s\n", key, txID)
}
jsonKeys, err := json.Marshal(keys)
if err != nil {
 return shim.Error(fmt.Sprintf("query operation failed. Error marshaling
 JSON: %s", err))
}
return shim.Success(jsonKeys)
}

```

## 12.10 marbles02

marbles02 是一个大理石资产管理的案例，介绍了如何在智能合约中定义资产，在区块链上进行资产的创建、转移、查询等操作。

chaincode 代码分析如下：

```
type marble struct {
 ObjectType string `json:"docType"` //docType is used to distinguish the
 various types of objects in state database
 Name string `json:"name"` //the fieldtags are needed to keep case
 from bouncing around
 Color string `json:"color"`
 Size int `json:"size"`
 Owner string `json:"owner"`
}
```

该智能合约定义了一个大理石的结构体，用于存储相关实体的信息，包括了类型、名称、颜色、尺寸、拥有者。

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 fmt.Println("invoke is running " + function)

 // Handle different functions
 if function == "initMarble" { //create a new marble
 return t.initMarble(stub, args)
 } else if function == "transferMarble" { //change owner of a specific marble
 return t.transferMarble(stub, args)
 } else if function == "transferMarblesBasedOnColor" { //transfer all marbles
 of a certain color
 return t.transferMarblesBasedOnColor(stub, args)
 } else if function == "delete" { //delete a marble
 return t.delete(stub, args)
 } else if function == "readMarble" { //read a marble
 return t.readMarble(stub, args)
 } else if function == "queryMarblesByOwner" { //find marbles for owner x
 using rich query
 return t.queryMarblesByOwner(stub, args)
 } else if function == "queryMarbles" { //find marbles based on an ad hoc rich query
 return t.queryMarbles(stub, args)
 } else if function == "getHistoryForMarble" { //get history of values for a marble
 return t.getHistoryForMarble(stub, args)
 } else if function == "getMarblesByRange" { //get marbles based on range query
 return t.getMarblesByRange(stub, args)
 }
 fmt.Println("invoke did not find func: " + function) //error
 return shim.Error("Received unknown function invocation")
}
```



该智能合约主要包括了 9 个方法：

## 1. initMarble 方法

创建一个大理石信息，并写入账本：

```
func (t *SimpleChaincode) initMarble(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
 var err error
 if len(args) != 4 {
 return shim.Error("Incorrect number of arguments. Expecting 4")
 }
 fmt.Println("- start init marble")
 if len(args[0]) <= 0 {
 return shim.Error("1st argument must be a non-empty string")
 }
 if len(args[1]) <= 0 {
 return shim.Error("2nd argument must be a non-empty string")
 }
 if len(args[2]) <= 0 {
 return shim.Error("3rd argument must be a non-empty string")
 }
 if len(args[3]) <= 0 {
 return shim.Error("4th argument must be a non-empty string")
 }
 marbleName := args[0]
 color := strings.ToLower(args[1])
 owner := strings.ToLower(args[3])
 size, err := strconv.Atoi(args[2])
 if err != nil {
 return shim.Error("3rd argument must be a numeric string")
 }
 marbleAsBytes, err := stub.GetState(marbleName)
 if err != nil {
 return shim.Error("Failed to get marble: " + err.Error())
 } else if marbleAsBytes != nil {
 fmt.Println("This marble already exists: " + marbleName)
 return shim.Error("This marble already exists: " + marbleName)
 }
 objectType := "marble"
 marble := &marble{objectType, marbleName, color, size, owner}
 marbleJSONasBytes, err := json.Marshal(marble)
 if err != nil {
 return shim.Error(err.Error())
 }
 err = stub.PutState(marbleName, marbleJSONasBytes)
 if err != nil {
 return shim.Error(err.Error())
 }
 indexName := "color~name"
 colorNameIndexKey, err := stub.CreateCompositeKey(indexName, []string{marble.
```

```

 Color, marble.Name))
 if err != nil {
 return shim.Error(err.Error())
 }
 value := []byte{0x00}
 stub.PutState(colorNameIndexKey, value)
 fmt.Println("- end init marble")
 return shim.Success(nil)
}

```

首先，该方法会进行重复性检查，查看是否有同名的已存在的大理石。如果没有，则将其序列化并存入账本。同时为了实现基于颜色的查询功能，使用 `CreateCompositeKey` 创建了组合键。

## 2. transferMarble 方法

改变大理石的所有者：

```

func (t *SimpleChaincode) transferMarble(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
 if len(args) < 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
 }
 marbleName := args[0]
 newOwner := strings.ToLower(args[1])
 fmt.Println("- start transferMarble ", marbleName, newOwner)
 marbleAsBytes, err := stub.GetState(marbleName)
 if err != nil {
 return shim.Error("Failed to get marble:" + err.Error())
 } else if marbleAsBytes == nil {
 return shim.Error("Marble does not exist")
 }
 marbleToTransfer := marble{}
 err = json.Unmarshal(marbleAsBytes, &marbleToTransfer) //unmarshal it aka
JSON.parse()
 if err != nil {
 return shim.Error(err.Error())
 }
 marbleToTransfer.Owner = newOwner //change the owner
 marbleJSONAsBytes, _ := json.Marshal(marbleToTransfer)
 err = stub.PutState(marbleName, marbleJSONAsBytes) //rewrite the marble
 if err != nil {
 return shim.Error(err.Error())
 }
 fmt.Println("- end transferMarble (success)")
 return shim.Success(nil)
}

```

该方法接受 2 个参数：大理石的名称以及新的所有者，将大理石的拥有者信息进行修改后存入账本。

### 3. transferMarblesBasedOnColor 方法

改变基于颜色，改变某种颜色下的所有大理石的所有者：

```
func (t *SimpleChaincode) transferMarblesBasedOnColor(stub shim.ChaincodeStubInterface, args []string) pb.Response {
 if len(args) < 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
 }
 color := args[0]
 newOwner := strings.ToLower(args[1])
 fmt.Println("- start transferMarblesBasedOnColor ", color, newOwner)
 coloredMarbleResultsIterator, err := stub.GetStateByPartialCompositeKey("color-name", []string{color})
 if err != nil {
 return shim.Error(err.Error())
 }
 defer coloredMarbleResultsIterator.Close()
 var i int
 for i = 0; coloredMarbleResultsIterator.HasNext(); i++ {
 responseRange, err := coloredMarbleResultsIterator.Next()
 if err != nil {
 return shim.Error(err.Error())
 }
 objectType, compositeKeyParts, err := stub.SplitCompositeKey(responseRange.Key)
 if err != nil {
 return shim.Error(err.Error())
 }
 returnedColor := compositeKeyParts[0]
 returnedMarbleName := compositeKeyParts[1]
 fmt.Printf("- found a marble from index:%s color:%s name:%s\n", objectType, returnedColor, returnedMarbleName)
 response := t.transferMarble(stub, []string{returnedMarbleName, newOwner})
 if response.Status != shim.OK {
 return shim.Error("Transfer failed: " + response.Message)
 }
 }
 responsePayload := fmt.Sprintf("Transferred %d %s marbles to %s", i, color, newOwner)
 fmt.Println("- end transferMarblesBasedOnColor: " + responsePayload)
 return shim.Success([]byte(responsePayload))
}
```

该方法用于将指定颜色的大理石的拥有者全部替换为新的用户。在这里用到了之前通过 init 创建的组合键，快速定位某种颜色对应的大理石，进行拥有者的修改。

### 4. delete 方法

用于删除大理石信息：

```
func (t *SimpleChaincode) delete(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
```



```

var jsonResp string
var marbleJSON marble
if len(args) != 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
}
marbleName := args[0]
valAsbytes, err := stub.GetState(marbleName) //get the marble from chaincode state
if err != nil {
 jsonResp = "{\"Error\":\"Failed to get state for " + marbleName + "\"}"
 return shim.Error(jsonResp)
} else if valAsbytes == nil {
 jsonResp = "{\"Error\":\"Marble does not exist: " + marbleName + "\"}"
 return shim.Error(jsonResp)
}
err = json.Unmarshal([]byte(valAsbytes), &marbleJSON)
if err != nil {
 jsonResp = "{\"Error\":\"Failed to decode JSON of: " + marbleName + "\"}"
 return shim.Error(jsonResp)
}
err = stub.DelState(marbleName) //remove the marble from chaincode state
if err != nil {
 return shim.Error("Failed to delete state:" + err.Error())
}
indexName := "color~name"
colorNameIndexKey, err := stub.CreateCompositeKey(indexName, []string{
 marbleJSON.Color, marbleJSON.Name})
if err != nil {
 return shim.Error(err.Error())
}
err = stub.DelState(colorNameIndexKey)
if err != nil {
 return shim.Error("Failed to delete state:" + err.Error())
}
return shim.Success(nil)
}

```

该方法基于传入的大理石名称进行删除操作，同时还需要对之前写入的组合键信息进行删除。

## 5. readMarble 方法

从账本中读取一个大理石的信息：

```

func (t *SimpleChaincode) readMarble(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
 var name, jsonResp string
 var err error
 if len(args) != 1 {
 return shim.Error("Incorrect number of arguments. Expecting name of the
 marble to query")
 }
}

```

```

name = args[0]
valAsBytes, err := stub.GetState(name)
if err != nil {
 jsonResp = "{\"Error\": \"Failed to get state for \" + name + \"\"}"
 return shim.Error(jsonResp)
} else if valAsBytes == nil {
 jsonResp = "{\"Error\": \"Marble does not exist: \" + name + \"\"}"
 return shim.Error(jsonResp)
}
return shim.Success(valAsBytes)
}

```

根据传入的大理石的名字进行大理石信息的读取。

## 6. queryMarblesByOwner 方法

根据传入的大理石的拥有者，查询对应的大理石信息：

```

func (t *SimpleChaincode) queryMarblesByOwner(stub shim.ChaincodeStubInterface,
args []string) pb.Response {
 if len(args) < 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
 }
 owner := strings.ToLower(args[0])
 queryString :=
fmt.Sprintf("{\"selector\":{\"docType\":\"marble\",\"owner\":\"%s\"}}", owner)
 queryResults, err := getQueryResultForQueryString(stub, queryString)
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(queryResults)
}

```

基于 couchdb 提供的富查询的特性，利用大理石的拥有者查询大理石的信息。

## 7. queryMarbles 方法

利用富查询查询大理石信息：

```

func (t *SimpleChaincode) queryMarbles(stub shim.ChaincodeStubInterface, args []
string) pb.Response {
 if len(args) < 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
 }
 queryString := args[0]
 queryResults, err := getQueryResultForQueryString(stub, queryString)
 if err != nil {
 return shim.Error(err.Error())
 }
 return shim.Success(queryResults)
}

```

根据参数传入的条件查询语句进行富查询操作。

## 8. getHistoryForMarble 方法

查询某个大理石的历史信息：

```
func (t *SimpleChaincode) getHistoryForMarble(stub shim.ChaincodeStubInterface,
 args []string) pb.Response {
 if len(args) < 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
 }
 marbleName := args[0]
 fmt.Printf("- start getHistoryForMarble: %s\n", marbleName)
 resultsIterator, err := stub.GetHistoryForKey(marbleName)
 if err != nil {
 return shim.Error(err.Error())
 }
 defer resultsIterator.Close()
 var buffer bytes.Buffer
 buffer.WriteString("[")
 bArrayMemberAlreadyWritten := false
 for resultsIterator.HasNext() {
 response, err := resultsIterator.Next()
 if err != nil {
 return shim.Error(err.Error())
 }
 if bArrayMemberAlreadyWritten == true {
 buffer.WriteString(",")
 }
 buffer.WriteString("{\"TxId\":")
 buffer.WriteString("\"")
 buffer.WriteString(response.TxId)
 buffer.WriteString("\"")
 buffer.WriteString(", \"Value\":")
 if response.IsDelete {
 buffer.WriteString("null")
 } else {
 buffer.WriteString(string(response.Value))
 }
 buffer.WriteString(", \"Timestamp\":")
 buffer.WriteString("\"")
 buffer.WriteString(time.Unix(response.Timestamp.Seconds, int64(response.
 Timestamp.Nanos)).String())
 buffer.WriteString("\"")
 buffer.WriteString(", \"IsDelete\":")
 buffer.WriteString("\"")
 buffer.WriteString(strconv.FormatBool(response.IsDelete))
 buffer.WriteString("\"")
 buffer.WriteString("}")
 bArrayMemberAlreadyWritten = true
 }
 buffer.WriteString("]")
}
```





```
fmt.Printf("- getHistoryForMarble returning:\n%s\n", buffer.String())
return shim.Success(buffer.Bytes())
}
```

根据大理石的名字利用 `GetHistoryForKey` 从历史数据库中查询大理石相关的历史信息  
的变动。

## 9. getMarblesByRange 方法

查询制定字典序范围内的大理石信息：

```
func (t *SimpleChaincode) getMarblesByRange(stub shim.ChaincodeStubInterface,
args []string) pb.Response {
 if len(args) < 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
 }
 startKey := args[0]
 endKey := args[1]
 resultsIterator, err := stub.GetStateByRange(startKey, endKey)
 if err != nil {
 return shim.Error(err.Error())
 }
 defer resultsIterator.Close()
 var buffer bytes.Buffer
 buffer.WriteString("[")
 bArrayMemberAlreadyWritten := false
 for resultsIterator.HasNext() {
 queryResponse, err := resultsIterator.Next()
 if err != nil {
 return shim.Error(err.Error())
 }
 if bArrayMemberAlreadyWritten == true {
 buffer.WriteString(",")
 }
 buffer.WriteString("{\"Key\":")
 buffer.WriteString("\"")
 buffer.WriteString(queryResponse.Key)
 buffer.WriteString("\"")
 buffer.WriteString(", \"Record\":")
 buffer.WriteString(string(queryResponse.Value))
 buffer.WriteString("}")
 bArrayMemberAlreadyWritten = true
 }
 buffer.WriteString("]")
 fmt.Printf("- getMarblesByRange queryResult:\n%s\n", buffer.String())
 return shim.Success(buffer.Bytes())
}
```

基于 `GetStateByRange` 方法对大理石从 `startKey` 到 `endKey` 进行检索，查询该范围内所有的大理石信息。



## 12.11 passthru

passthru 链码主要示范了跨链码调用, 提供 ChaincodeID、function、args 即可进行跨链码的操作。

chaincode 代码分析如下:

```
func (p *PassthruChaincode) iq(stub shim.ChaincodeStubInterface, function string,
 args []string) pb.Response {
 if function == "" {
 return shim.Error("Chaincode ID not provided")
 }
 chaincodeID := function
 return stub.InvokeChaincode(chaincodeID, toChaincodeArgs(args...), "")
}
func (p *PassthruChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 return p.iq(stub, function, args)
}
```

该链码主要基于 stub.InvokeChaincode 进行一个跨链码的调用操作, 只需 ChaincodeID、function、args 即可, 如需指明 channelName 可在最后指定。

## 12.12 sleeper

sleeper 链码主要是用于为测试 chaincode.executetimeout 属性而设计的合约, 能够根据传入的 sleeptime 来验证超时的时间。

chaincode 代码分析如下:

```
type SleeperChaincode struct {
}

func (t *SleeperChaincode) sleep(sleepTime string) {
 st, _ := strconv.Atoi(sleepTime)
 if st >= 0 {
 time.Sleep(time.Duration(st) * time.Millisecond)
 }
}

func (t *SleeperChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
 args := stub.GetStringArgs()
 if len(args) != 1 {
 return shim.Error("Incorrect number of arguments. Expecting 1")
 }
 sleepTime := args[0]
 t.sleep(sleepTime)
 return shim.Success(nil)
}
```



```

func (t *SleeperChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
 if function == "put" {
 if len(args) != 3 {
 return shim.Error("Incorrect number of arguments. Expecting 3")
 }
 return t.invoke(stub, args)
 } else if function == "get" {
 if len(args) != 2 {
 return shim.Error("Incorrect number of arguments. Expecting 2")
 }
 return t.query(stub, args)
 }
 return shim.Error("Invalid invoke function name. Expecting \"put\" or \"get\"")
}

func (t *SleeperChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {
 key := args[0]
 val := args[1]
 err := stub.PutState(key, []byte(val))
 if err != nil {
 return shim.Error(err.Error())
 }
 sleepTime := args[2]
 t.sleep(sleepTime)
 return shim.Success([]byte("OK"))
}

func (t *SleeperChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
 key := args[0]
 val, err := stub.GetState(key)
 if err != nil {
 return shim.Error(err.Error())
 }
 sleepTime := args[1]
 t.sleep(sleepTime)
 return shim.Success(val)
}

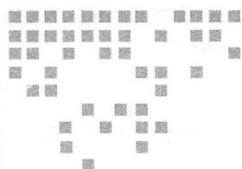
func main() {
 err := shim.Start(new(SleeperChaincode))
 if err != nil {
 fmt.Printf("Error starting Sleeper chaincode: %s", err)
 }
}

```

该链码在 Init、Invoke 中均传入了 sleepTime，用于进行睡眠操作，可设置不同的时间以测试配置中 chaincode.executetimeout 字段是否生效。







## Chapter 13 第 13 章

# Fabric-samples 项目分析与实践

本章将通过项目实例来帮助读者更好地理解 Fabric，提升实战经验。

## 13.1 Fabric-samples 项目结构

本节将介绍 Fabric-samples 项目结构。

请先访问 Prerequisites 网页并确保计算机上已经安装了必需的依赖项。

选择 clone fabric-samples 的工作目录，运行 clone 命令，并进入 fabric-samples 目录：

```
git clone https://github.com/hyperledger/fabric-samples.git
cd fabric-samples
```

- fabric-samples - 目录主要包含了示例项目的代码。在该目录下运行 ls 命令，会看到以下内容：

```
LICENSE bin fabric-ca MAINTAINERS.md chaincode first-network
README.md chaincode-docker-devmode high-throughput balance-transfer
config scripts basic-network fabcar
```

下面是一个关于 Fabric-samples 项目结构的简要说明：

- ❑ first-network 是一个由两个组织组成的示例 Fabric 网络，每个组织持有 2 个 peer 节点，以及一个 solo 排序服务。网络启动后会自动完成一笔转账。
- ❑ basic-network 建立了开发 chaincode 和基本应用程序所需的最小节点数量。它只有一个 peer，因此也只有一个 organization。fabcar 应用的建立基于这个网络。
- ❑ fabcar 是一个基于区块链网络的应用程序，为用户提供查询账本（包含特定记录）以及更新账本（添加记录）的功能。



- ❑ high-throughput 用来了解如何在每秒处理数千次（在账本中更新同一资产）事务时正确设计 chaincode 数据模型。
- ❑ balance-transfer 是一个示例 Node.js 的应用程序，用来演示 fabric-client 和 fabric-ca-client Node.js SDK API。
- ❑ fabric-ca 样例演示了如何使用 Fabric CA 客户端和服务端来生成所有加密材料以及如何使用基于属性的访问控制。
- ❑ chaincode 链码程序样例。链码运行在与背书节点相隔离的安全的 Docker 容器中。
- ❑ chaincode-docker-devmode 链码在 Docker 内的开发者模式样例。在“开发模式”中，链码中用户构建和启动。

完成本章的学习之后，你应该会了解 fabric-samples 示例项目的功能，接下来的章节我们将具体讲述它们。

## 13.2 First-network



**注意** 在之后的操作步骤中，这些说明已经被验证，适用于被标记为“1.0.0-rc1”的 Docker 镜像和提供 tar 文件中的预编译的实用程序。如果你在当前的主分支下使用下列命令以及镜像或者工具，你可能会看到一些配置和 panic 错误。

构建你的第一个网络（BYFN）场景提供了由两个组织组成的示例 Hyperledger Fabric 网络，每个组织持有 2 个 peer 节点，以及一个 solo 排序服务。

### 13.2.1 安装预置环境

在我们开始之前，你需要检查一下在你开发区块链应用程序或者在 Hyperledger Fabric 平台上是否已经安装了预置环境。

你还需要下载并安装 Hyperledger Fabric Samples。你会注意到 fabric-samples 文件夹中包含了许多示例。我们将使用 first-network 这个例子。现在让我们打开这个子目录。

```
cd first-network
```



**注意** 本节中提供的命令必须运行在 fabric-network 的子目录 first-network 中。如果你选择从其他位置运行命令，本节提供的一些列脚本将无法找到对应的二进制。

### 13.2.2 想要现在运行吗？

我们提供一个完全注释的脚本 byfn.sh，利用脚本里这些 Docker 镜像可以快速引导一个有 4 个代表、2 个不同组织的 peer 节点以及一个排序服务节点的 Hyperledger



fabric 网络。脚本还将启动一个容器来运行一个将 peer 节点以加入 channel、部署实例化链码服务以及驱动已经部署的链码执行交易的脚本。

以下是该 byfn.sh 脚本的帮助文档：

```
./byfn.sh -h
Usage:
 byfn.sh -m up|down|restart|generate [-c <channel name>] [-t <timeout>]
 byfn.sh -h|--help (print this message)
 -m <mode> - one of 'up', 'down', 'restart' or 'generate'
 - 'up' - bring up the network with docker-compose up
 - 'down' - bring up the network with docker-compose up
 - 'restart' - bring up the network with docker-compose up
 - 'generate' - generate required certificates and genesis block
 -c <channel name> - config name to use (defaults to "mychannel")
 -t <timeout> - CLI timeout duration in microseconds (defaults to 10000)
```

Typically, one would first generate the required certificates and genesis block, then bring up the network. e.g.:

```
byfn.sh -m generate -c <channelname>
byfn.sh -m up -c <channelname>
```

如果你选择不提供 channel 名称，则脚本将使用默认名称 mychannel。CLI 超时参数（用 -t 标志指定）是一个可选值；如果你选择不设置它，那么 CLI 容器将会在脚本执行完之后退出。

### 13.2.3 生成网络神器

执行以下命令，你将会看到会发生什么。根据 yes/no 命令行提示的简要说明。输入 y 来执行描述的动作。

```
./byfn.sh -m generate
Generating certs and genesis block for with channel 'mychannel' and CLI timeout
of '10000'
Continue (y/n)?y
proceeding ...
/Users/xxx/dev/fabric-samples/bin/cryptogen

#####
Generate certificates using cryptogen tool
#####
org1.example.com
2017-06-12 21:01:37.334 EDT [bccsp] GetDefault -> WARN 001 Before using BCCSP,
please call InitFactories(). Falling back to bootBCCSP.
...

/Users/xxx/dev/fabric-samples/bin/configtxgen
#####
Generating Orderer Genesis block
```





```
#####
2017-06-12 21:01:37.558 EDT [common/configtx/tool] main -> INFO 001 Loading
configuration
2017-06-12 21:01:37.562 EDT [msp] getMspConfig -> INFO 002 intermediate certs
folder not found at [/Users/xxx/dev/byfn/crypto-config/ordererOrganizations/
example.com/msp/intermediatecerts]. Skipping.: [stat /Users/xxx/dev/byfn/
crypto-config/ordererOrganizations/example.com/msp/intermediatecerts: no such
file or directory]
...
2017-06-12 21:01:37.588 EDT [common/configtx/tool] doOutputBlock -> INFO 00b
Generating genesis block
2017-06-12 21:01:37.590 EDT [common/configtx/tool] doOutputBlock -> INFO 00c
Writing genesis block

#####
Generating channel configuration transaction 'channel.tx'
#####
2017-06-12 21:01:37.634 EDT [common/configtx/tool] main -> INFO 001 Loading
configuration
2017-06-12 21:01:37.644 EDT [common/configtx/tool] doOutputChannelCreateTx ->
INFO 002 Generating new channel configtx
2017-06-12 21:01:37.645 EDT [common/configtx/tool] doOutputChannelCreateTx ->
INFO 003 Writing new channel tx

#####
Generating anchor peer update for Org1MSP
#####
2017-06-12 21:01:37.674 EDT [common/configtx/tool] main -> INFO 001 Loading
configuration
2017-06-12 21:01:37.678 EDT [common/configtx/tool] doOutputAnchorPeersUpdate ->
INFO 002 Generating anchor peer update
2017-06-12 21:01:37.679 EDT [common/configtx/tool] doOutputAnchorPeersUpdate ->
INFO 003 Writing anchor peer update

#####
Generating anchor peer update for Org2MSP
#####
2017-06-12 21:01:37.700 EDT [common/configtx/tool] main -> INFO 001 Loading
configuration
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate ->
INFO 002 Generating anchor peer update
2017-06-12 21:01:37.704 EDT [common/configtx/tool] doOutputAnchorPeersUpdate ->
INFO 003 Writing anchor peer update
```

第一步生成我们各种网络实体的所有证书和密钥，genesis block 用于引导排序服务，以及配置 Channel 所需要的一组交易配置集合。

### 13.2.4 启动网络

接下来，你可以使用以下命令来启动整个网络。再次提示你是否继续。回答 y:



```

./byfn.sh -m up
Starting with channel 'mychannel' and CLI timeout of '10000'
Continue (y/n)?y
proceeding ...
Creating network "net_byfn" with the default driver
Creating peer0.org1.example.com
Creating peer1.org1.example.com
Creating peer0.org2.example.com
Creating orderer.example.com
Creating peer1.org2.example.com
Creating cli

```

```

 ____ ____ ____ ____ ____
 / _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|

```

Channel name : mychannel

Creating channel...

The logs will continue from there. This will launch all of the containers, and then drive a complete end-to-end application scenario. Upon successful completion, it should report the following in your terminal window:

日志将继续。然后启动所有容器，驱动一个端到端的应用场景。成功以后，在终端窗口中会报告以下内容：

```

2017-05-16 17:08:01.366 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing
local MSP
2017-05-16 17:08:01.366 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining
default signing identity
2017-05-16 17:08:01.366 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AB
1070A6708031A0C08F1E3ECC80510...6D7963631A0A0A0571756572790A0161
2017-05-16 17:08:01.367 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: E61DB3
7F4E8B0D32C9FE10E3936BA9B8CD278FAA1F3320B08712164248285C54
Query Result: 90
2017-05-16 17:08:15.158 UTC [main] main -> INFO 008 Exiting.....
===== Query on PEER3 on channel 'mychannel' is successful
=====

===== All GOOD, BYFN execution completed =====

```

```

 ____ ____ ____ ____ ____
 / _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|
| _ \| _ \| _ \| _ \| _ \|

```

你可以滚动这些日志去查看各种交易。



### 13.2.5 关闭网络

最后，让我们把脚本全部停下来，这样我们可以一步一步地探索网络设置。以下操作将关闭你的容器，移除加密材料和 4 个配置信息，并且从 Docker 仓库删除 chaincode 镜像。你将再一次被提示是否继续，回答 y:

```
./byfn.sh -m down
Stopping with channel 'mychannel' and CLI timeout of '10000'
Continue (y/n)?y
proceeding ...
WARNING: The CHANNEL_NAME variable is not set. Defaulting to a blank string.
WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.
Removing network net_byfn
468aaa6201ed
...
Untagged: dev-peer1.org2.example.com-mycc-1.0:latest
Deleted: sha256:ed3230614e64e1c83e510c0c282e982d2b06d148b1c498bbdcc429e2b2531e91
...
```

在接下来的章节中，我们将介绍构建功能齐全的 Hyperledger fabric 网络的各种要求和步骤。

### 13.2.6 加密生成器

我们将使用 cryptogen 工具为我们生成各种网络实体的加密材料 (x.509 证书)。这些加密材料是身份的代表，在我们的网络实体进行交流和交易时这些材料用于进行签名 / 验证身份。

Cryptogen 消费一个包含网络拓扑的 crypto-config.yaml，并允许我们为组织和属于这些组织的组件生成一组证书和密钥。每个组织都配置了唯一的根证书 (ca-cert)，它将特定组件 (peers 和 orders) 绑定到该组织。通过为每一个组织分配唯一的 CA 证书，我们正在模仿一个经典的网络，这个网络中的成员将使用自己的证书颁发机构。Hyperledger Fabric 中的交易和通信是通过存储在 keystore 中的实体的私钥签名，然后通过公钥手段进行验证 (signcerts) 的。

在这个文件里有一个 count 变量，我们将使用它来指定每个组织中 peer 的数量；在我们的例子中，每个组织有两个 peer。我们现在不会深入研究 x.509 证书和公钥基础设施的细节。如果你有兴趣，可以自己学习。

在运行该工具之前，让我们快速浏览一下这段代码 crypto-config.yaml。特别注意在 OrdererOrgs 头下的 Name、Domain 和 Specs 参数：

```
OrdererOrgs:
#-----
Orderer

- Name: Orderer
 Domain: example.com
```





```

"Specs" - See PeerOrgs below for complete description

Specs:
 - Hostname: orderer

"PeerOrgs" - Definition of organizations managing peer nodes

PeerOrgs:

Org1

- Name: Org1
 Domain: org1.example.com

```

网络实体的命名约定如下: "{.Hostname}},{.Domain}"。所以使用我们的排序节点作为参考点,它与 Order 的 MSP ID 相关联。该文件包含了有关定义和语法的大量文档。你还可以参考 Membership Service Providers(MSP),以便更深入地了解 MSP。

我们运行 cryptogen 工具,生成的证书和密钥将被保存到名为 crypto-config 的文件夹中。

### 13.2.7 配置交易生成器

configtxgen 工具用于创建 4 个配置工作: order 的 genesis block, channel 的 channel configuration transaction, \* 以及两个 anchor peer transactions (一个对应一个 Peer 组织)。

有关此工具的完整说明,请参阅 Channel Configuration(configtxgen)。

order block 是一个 ordering service 的创世区块, channel transaction 文件在 Channel 创建的时候广播给 order。anchor peer transactions, 正如名称所示,指定了每个组织在此 channel 上的 Anchor peer。

Configtxgen 使用一个包含示例网络的 configtx.yaml 文件。具有 3 个成员: 一个排序服务组织 OrdererOrg 以及两个节点组织 (Org1&Org2)。每个组织管理和持有 2 个 peer 节点。该文件还指定了一个 SampleConsortium 的联盟,由上述两个节点组织构成。请特别注意此文件顶部的 "Profiles" 部分,这里有两个独特的标题: 一个是 orderer 的创世区块 -TwoOrgsOrdererGenesis; 另一个是针对管道的 TwoOrgsChannel。

这些标题很重要,因为在我们创建我们的工作的时候它们将作为传递的参数。



**注意** 我们的 SampleConsortium 在系统界别的配置文件中定义,然后由渠道级别配置文件引用。管道存在于联盟的范围内,所有的联盟必须定义在整个网络范围内。

此文件还包含两个值得注意的附加参数: 首先,我们为每个组织指定了锚点节点 (peer0.org1.example.com 和 peer0.org2.example.com); 其次,我们为每个成

员指定 MSP 文件夹，用来存储每个组织在 orderer genesis block 中指定的根证书。这是一个关键的概念。现在任意和 ordering service 通信的网络实体都可以对其数字签名进行验证。

### 13.2.8 运行工具

你可以用 configtxgen 和 cryptogen 命令来手动生成证书 / 密钥和各种配置文件。你也可以尝试使用 byfn.sh 脚本来完成你的目标。

必要的话，你可以参考 byfn.sh 脚本中的 generateCerts 函数去生成定义在 crypto-config.yaml 文件中用于你的网络配置的相关证书。然而，为了方便起见，这里我们也会进行相应介绍。

首先，我们来运行 cryptogen 这个工具。我们的二进制文件在 bin 目录中，所以我们需要提供工具所在的相对路径。

```
../bin/cryptogen generate --config=./crypto-config.yaml
```

你可能会看到以下警告，这是无害的，请忽略它：

```
[bccsp] GetDefault -> WARN 001 Before using BCCSP, please call InitFactories().
Falling back to bootBCCSP.
```

接下来，我们需要告诉 configtxgen 工具需要提取的 configtx.yaml 所在的位置。我们会告诉它在我们当前所在工作目录。

首先，我们需要设置一个环境变量来告诉 configtxgen 去哪里寻找 configtx.yaml。然后，我们将调用 configtxgen 工具去创建 orderer genesis block：

```
export FABRIC_CFG_PATH=$PWD
../bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block
```

你可以忽略有关中间证书、证书撤销列表 (crls) 和 MSP 配置的日志警告。我们没有在示例网络中使用其中的任何一个。

接下来，我们需要创建 channel transaction 配置。请确保替换 \$CHANNEL\_NAME 或者将 CHANNEL\_NAME 设置为整个说明中可以使用的环境变量：

```
export CHANNEL_NAME=mychannel

this file contains the definitions for our sample channel
../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL_NAME
```

接下来，我们将在正在构建的通道上定义 Org1 的 anchor peer。请再次确认 \$CHANNEL\_NAME 已被替换或者为以下命令设置了环境变量：

```
../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
```

现在，我们将在同一个通道定义 Org2 的 anchor peer：

```
../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-
artifacts/Org2MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

### 13.2.9 启动网络

我们将利用 docker-compose 脚本来启动我们的区块链网络。docker-compose 文件利用我们之前下载的镜像，并用以前生成的 genesis.block 来引导 orderer。

```
working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME}; sleep $TIMEOUT'
volumes
```

如果没有注释，该脚本将在网络启动时执行所有命令，正如我们在幕后发生的情况中所描述的那样。然而，我们想手动执行命令，以便公开每个调用的语法和功能。

适当地为 TIMEOUT 传递较高的值（以秒为单位）；默认情况下 CLI 容器将在 60 秒之后退出。

启动你的网络：

```
CHANNEL_NAME=$CHANNEL_NAME TIMEOUT=<pick_a_value> docker-compose -f docker-
compose-cli.yaml up -d
```

如果要实时查看你的区块链网络的日志，请不要提供 -d 标志。如果你需要日志流，你需要打开第二个终端来执行 CLI 命令。

#### 1. 环境变量

为了使针对 peer0.org1.example.com 的 CLI 命令起作用，我们需要使用下面给出 4 个环境变量来介绍我们的命令。peer0.org1.example.com 涉及的这些变量将被拷贝到 CLI 容器中，因此我们不需要复制它们。然而，如果你发送调用请求到其他的 peer 节点或者 orderer，则需要相应地提供这些值。检查 docker-compose-base.yaml 中的具体路径：

```
Environment variables for PEER0

CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

#### 2. 创建 & 加入信道

我们将使用 docker exec 命令进入 CLI 容器：

```
docker exec -it cli bash
```



命令运行如果成功，你将会看到下列信息：

```
root@0d78bb69300d:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

我们使用 `configtxgen` 工具生成信道配置 `-channel.tx`，并将这个配置作为请求的一部分传递给 `order`。



**注意** `--cafile` 会作为命令的一部分。这是 `orderer` 的 `root cert` 的本地路径，允许我们去验证 TLS 握手。

我们使用 `-c` 标志指定 `channel` 的名字，使用 `-f` 标志指定配置交易。在这个例子中配置文件名是 `channel.tx`，当然你也可以使用不同的名称来挂载你自己的交易配置。

```
export CHANNEL_NAME=mychannel

the channel.tx file is mounted in the channel-artifacts directory within your
 CLI container
as a result, we pass the full path for the file
we also pass the path for the orderer ca-cert in order to verify the TLS
 handshake
be sure to replace the $CHANNEL_NAME variable appropriately

peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-
artifacts/channel.tx --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/
github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/
orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

此命令返回一个创世区块 `<channel-ID.block>`，我们将使用它加入信道。它包含了 `channel.tx` 中的配置信息。



**注意** 剩下的命令将会留在 CLI 容器内执行。当目标节点是除了 `peer0.org1.example.com` 以外的节点时，你必须记住所有的命令必须在相应的环境变量下执行。

现在让我们加入 `peer0.org1.example.com` 频道。

```
By default, this joins ``peer0.org1.example.com`` only
the <channel-ID>.block was returned by the previous command

peer channel join -b <channel-ID.block>
```

你可以修改 4 个环境变量来让别的节点加入信道。

### 3. 安装和实例化链码

本节我们将利用一个现有的简单链码，来学习如何编写自己的链码。

应用程序和区块链账本会通过 `chaincode` 相互影响。因此，我们需要在每个会执行以及背书我们交易的 `peer` 节点上安装 `chaincode`，然后在信道上实例化 `chaincode`。

首先，将示例代码安装到 4 个 `peer` 节点中的一个上。下面这个命令将源代码放到 `peer`

节点的文件系统中。

```
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02
```

接下来，在信道上实例化 chaincode。这将初始化信道上的链码，设置链码的背书策略，为目标 peer 节点启动一个 chaincode 容器（注意 -P 参数）。我们需要指定的、当这个 chaincode 交易需要被验证的时候的背书策略。

在下面的命令中，我们指定 -P "OR ('Org0MSP.member', 'Org1MSP.member')" 作为背书策略。这意味着我们只需要 org1 或者 org2 组织中的一个节点的背书即可（即只有一个背书）。如果我们将语法变为 AND，那么我们就需要 2 个背书者。

```
be sure to replace the $CHANNEL_NAME environment variable
if you did not install your chaincode with a name of mycc, then modify that argument as well
peer chaincode instantiate -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

#### 4. 查询

让我们查询一下 a 的值，以确保链码被正确实例化、state DB 被填充。查询的语法如下：

```
be sure to set the -C and -n flags appropriately
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

#### 5. 调用

现在让我们从 a 账户转 10 元到 b 账户。这个交易将创建一个新的区块并更新 state DB。调用语法如下：

```
be sure to set the -C and -n flags appropriately
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

#### 6. 查询

让我们确认下我们之前的调用被正确执行了。我们初始化了 a 的值为 100，在上一次调用的时候转移了 10 元给 b。因此，查询 a 应该展示 90 元。查询的语法如下：

```
be sure to set the -C and -n flags appropriately
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

我们应该看到以下内容：

Query Result: 90

随时重新开始并操作键值对和随后的调用。

## 7. 幕后发生了什么？

注意：以下步骤描述了在 `script.sh` 脚本中没有注释掉的 `docker-compose-cli.yaml` 文件中的场景，其中使用了 `./byfn.sh -m down` 并确保了命令执行成功，然后使用相同的 `docker-compose` 提示去启动你的网络。

- ❑ `script.sh` 脚本被拷贝到 CLI 容器中。这个脚本驱动了 `createChannel` 命令此命令使用提供的 `channel name` 以及信道配置的 `channel.tx` 文件。
- ❑ `createChannel` 命令的产出是一个创世区块 `-<your_channel_name>.block-`，这个创世区块被存储在 `peer` 节点的文件系统中，同时包含了 `channel.tx` 的信道配置。
- ❑ `joinChannel` 命令被 4 个 `peer` 节点执行，作为之前产生的 `genesis block` 的输入。这个命令介绍了 `peer` 节点如何加入 `<your_channel_name>` 以及如何利用 `<your_channel_name>.block` 去创建一条链。
- ❑ 现在我们有由 4 个 `peer` 节点以及 2 个组织构成的信道。这是我们的 `TwoOrgs-Channel` 配置文件。
- ❑ `peer0.org1.example.com` 和 `peer1.org1.example.com` 属于 `Org1`；`peer0.org2.example.com` 和 `peer1.org2.example.com` 属于 `Org2`。
- ❑ 这些关系是通过 `crypto-config.yaml` 定义的，MSP 路径在 `docker-compose` 文件中被指定。
- ❑ `Org1MSP(peer0.org1.example.com)` 和 `Org2MSP(peer0.org2.example.com)` 的 `anchor peers` 将在后续被更新。我们通过携带 `channel` 的名字传递 `Org1-MSPanchors.tx` 和 `Org2MSPanchors.tx` 配置到排序服务来实现 `anchor peer` 的更新。
- ❑ 一个链码 `-chaincode_example02` 被安装在 `peer0.org1.example.com` 和 `peer0.org2.example.com` 中。
- ❑ 这个链码在 `peer0.org2.example.com` 中被实例化。实例化过程将链码添加到信道上，并启动 `peer` 节点对应的容器，同时初始化和链码服务有关的键值对。示例的初始化的值是 `["a", "100", "b", "200"]`。实例化的结果是一个名为 `dev-peer0.org2.example.com-mycc-1.0` 的容器启动了。
- ❑ 实例化过程同样为背书策略传递相关参数。策略被定义为 `-P "OR ('Org1MSP.member', 'Org2MSP.member')"`，意思是任何交易必须被 `Org1` 或者 `Org2` 背书。
- ❑ 一个针对 `a` 的查询发往 `peer0.org1.example.com`。链码服务已经被安装在了 `peer0.org1.example.com`，因此这次查询将启动一个名为 `dev-peer0`。





## 10. 如何查看链码日志？

检查每个独立的链码服务容器来查看每个容器内分隔的交易。下面是每个链码服务容器的日志的组合：

```
$ docker logs dev-peer0.org2.example.com-myc-1.0
04:30:45.947 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Init
Aval = 100, Bval = 200

$ docker logs dev-peer0.org1.example.com-myc-1.0
04:31:10.569 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"100"}
ex02 Invoke
Aval = 90, Bval = 210

$ docker logs dev-peer1.org2.example.com-myc-1.0
04:31:30.420 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"90"}
```

### 13.2.10 了解 Docker Compose 技术

BYFN 示例给我们提供了两种风格的 Docker Compose 文件，它们都继承自 `docker-compose-base.yaml`（base 目录下）。第一种风格——`docker-compose-cli.yaml` 给我们提供了 1 个 CLI 容器 1 个 orderer 容器和 4 个 peer 容器。我们用此文件来展开这个页面上的所有说明。



**注意** 本节剩余部分涵盖了为 SDK 设计的 `docker-compose` 文件。有关运行这些测试的详细信息，请参阅 Node SDK 仓库。

第二种风格是 `docker-compose-e2e.yaml`，被构造为使用 Node.js SDK 来运行端到端的测试。除了 SDK 的功能之外，它还运行 `fabric-ca` 服务的容器。因此，我们能够向组织的 CA 节点发送 REST 的请求以用于注册和登记。

如果你在没有运行 `byfn.sh` 脚本的情况下想使用 `docker-compose-e2e.yaml`，我们需要进行 4 个轻微的修改。我们需要指出本组织 CA 的私钥。你可以在 `crypto-config` 文件夹中找到这些值。举个例子，为了定位 `Org1` 的私钥，我们将使用 `crypto-config/peerOrganizations/org1.example.com/ca/`。`Org2` 的路径为 `crypto-config/peerOrganizations/org2.example.com/ca/`。

在 `docker-compose-e2e.yaml` 里为 `ca0` 和 `ca1` 更新 `FABRIC_CA_SERVER_TLS_KEYFILE` 变量。你同样需要编辑 `command` 去启动 `ca server` 的路径，并为每个 CA 容器提供了 2 次同样的私钥。

### 13.2.11 使用 CouchDB

状态数据库可以从默认的 `goleveldb` 切换到 `CouchDB`。链码同样能使用 `CouchDB`。但是, `CouchDB` 提供了额外的能力来根据 JSON 形式的链码服务数据提供更加丰富、复杂的查询。

使用 `CouchDB` 代替默认的数据库 (`goleveldb`), 除了在启动网络的时候传递 `docker-compose-couch.yaml` 之外, 请遵循前面提到的生成配置文件的过程:

```
CHANNEL_NAME=$CHANNEL_NAME TIMEOUT=<pick_a_value> docker-compose -f docker-
compose-cli.yaml -f docker-compose-couch.yaml up -d
```

`chaincode_example02` 现在应该使用下面的 `CouchDB`。



**注意** 如果你选择将 `fabric-couchdb` 容器端口映射到主机端口, 请确保你意识到了安全性的影响。在开发环境中映射端口可以使 `CouchDB` REST API 可用, 并允许通过 `CouchDB` Web 界面 (Fauxton) 对数据库进行可视化。生产环境将避免端口映射, 以限制对 `CouchDB` 容器的外部访问。

你可以使用上面列出的步骤通过 `CouchDB` 来执行 `chaincode_example02`, 然而为了执行 `CouchDB` 查询, 你将需要使用被格式化为 JSON 的数据 (例如 `marbles02`)。你可以在 `fabric/examples/chaincode/go` 目录中找到 `marbles02` 链码服务。

我们将按照上述创建和加入频道的过程创建和加入信道。一旦你将 `peer` 节点加入到了信道, 请使用以下步骤与 `marbles02` 链码交互。

在 `peer0.org1.example.com` 上安装和实例化链码:

```
be sure to modify the $CHANNEL_NAME variable accordingly for the instantiate
command
```

```
peer chaincode install -n marbles -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/marbles02
peer chaincode instantiate -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/
tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -v 1.0 -c
'{"Args":["init"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

创建一些 `marbles` 并移动它们:

```
be sure to modify the $CHANNEL_NAME variable accordingly
```

```
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c '{"Args":["initMarbl
e","marble1","blue","35","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
```



```

ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c '{"Args":["initMarbl
e","marble2","red","50","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c '{"Args":["initMarbl
e","marble3","blue","70","tom"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c '{"Args":["transferM
arble","marble2","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c '{"Args":["transferM
arblesBasedOnColor","blue","jerry"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/ordererOrganizations/example.com/orderers/orderer.example.com/
msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n marbles -c
 '{"Args":["delete","marble1"]}'

```

如果你选择在 docker-compose 文件中映射你的 CouchDB 端口，那么你现在就可以通过 CouchDB Web 界面 (Fauxton) 通过在浏览器导航中打开如下 URL: [http://localhost:5984/\\_utils](http://localhost:5984/_utils)。

你应该可以看到一个名为 mychannel (或者你的唯一的信道名字) 的数据库以及它的文档在里面:

Note

For the below commands, be sure to update the \$CHANNEL\_NAME variable appropriately.



对于下面的命令，请确定 \$CHANNEL\_NAME 变量被更新了。

你可以在 CLI 中运行常规的查询 (例如读取 marble2):

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["readMarble","marble2"]}'
```

marble2 的输出如下:

```
Query Result: {"color":"red","docType":"marble","name":"marble2","owner":"jerry",
,"size":50}
```

你可以检索特定 marble 的历史记录，例如 marble1:

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["getHistoryForMarb
le","marble1"]}'
```

输出在 marble1 的交易:

```
Query Result: [{"TxId":"1c3d3caf124c89f91a4c0f353723ac736c58155325f02890adebaa15
e16e6464", "Value":{"docType":"marble","name":"marble1","color":"blue","siz
e":35,"owner":"tom"}},{ "TxId":"755d55c281889eaeebf405586f9e25d71d36eb3d35420
af833a20a2f53a3eefd", "Value":{"docType":"marble","name":"marble1","color":"
blue","size":35,"owner":"jerry"}},{ "TxId":"819451032d813dde6247f85e56a892625
55e04f14788ee33e28b232eef36d98f", "Value":{}}]
```

你还可以对数据内容执行丰富的查询操作，例如通过拥有者 jerry 查询 marble：

```
peer chaincode query -C $CHANNEL_NAME -n marbles -c '{"Args":["queryMarblesByOwn
er","jerry"]}'
```

输出应该显示 2 个属于 jerry 的 marble：

```
Query Result: [{"Key":"marble2", "Record":{"color":"red","docType":"marble","nam
e":"marble2","owner":"jerry","size":50}},{ "Key":"marble3", "Record":{"color"
:"blue","docType":"marble","name":"marble3","owner":"jerry","size":70}}]
```

### 13.2.12 关于数据持久化的提示

如果需要在 peer 容器或者 CouchDB 容器进行数据持久化，一种选择是将 Docker 容器内相应的目录挂载到容器所在的宿主机的目录中。例如，你可以添加下面的代码到 docker-compose-base.yaml 文件下 peer 的约定中：

```
volumes:
 - /var/hyperledger/peer0:/var/hyperledger/production
```

对于 CouchDB 容器，你可以在 CouchDB 的约定中添加如下两行代码：

```
volumes:
 - /var/hyperledger/couchdb0:/opt/couchdb/data
```

### 13.2.13 故障排除

始终保持你的网络是全新的。使用以下命令来移除之前生成的 artifacts, crypto, containers 以及 chaincode images：

```
./byfn.sh -m down
```

如果你不移除容器和镜像，你将会看到错误信息。

如果你看到 Docker 相关的错误信息，则应检查你的版本（应为 1.12 或更高版本），然后重启你的 Docker 进程。Docker 的问题通常不会被立即识别。例如，你可能会看到由容器内加密材料导致的错误。

删除你的镜像，并从头开始：

```
docker rm -f $(docker ps -aq)
docker rmi -f $(docker images -q)
```

如果你发现你的创建、实例化、调用或者查询命令，请确保你已经更新了信道和链码

的名字。提供的示例命令中有占位符。

如果你看到如下错误，则说明可能以前运行的链码服务（例如 `dev-peer1.org2.example.com-mycc-1.0` 或 `dev-peer0.org1.example.com-mycc-1.0`）有问题：

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing
chaincode code mycc:1.0(chaincode /var/hyperledger/production/chaincodes/
mycc.1.0 exits)
```

删除这些链码服务，然后重试。

```
docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
```

如果你看到以下错误信息，则应确保你的 Fabric 网络运行在被标记为 `latest` 的 `1.0.0-rc1` 镜像上。

```
Error connecting: rpc error: code = 14 desc = grpc: RPC failed fast due to transport
failure
Error: rpc error: code = 14 desc = grpc: RPC failed fast due to transport failure
```

如果你看到了以下错误内容，那么说明没有正确设置 `FABRIC_CFG_PATH` 环境变量。`configtxgen` 工具需要这个变量才能找到 `configtx.yaml`。返回并执行 `export FABRIC_CFG_PATH=$PWD`，然后重新创建 `channel` 配置。

```
[configtx/tool/localconfig] Load -> CRIT 002 Error reading configuration:
Unsupported Config Type ""
panic: Error reading configuration: Unsupported Config Type ""
```

要清理网络，请使用 `down` 选项：

```
./byfn.sh -m down
```

如果你看到一条指示中依然有“`active endpoints`”，然后清理你的 Docker 网络。这将会清除你之前的网络并且给你一个全新的环境：

```
docker network prune
```

你将看到以下消息：

```
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

选择 `y`。

如果你仍旧看到了错误，请在 Hyperledger Rocket Chat 的 `# fabric-questions` 频道上分享你的日志。

## 13.3 basic-network

注意，此基本配置使用预先生成的证书和密钥材料，并且还具有预定义的操作来初始



化一个名为“mychannel”的通道。

要重新生成这个材料，只需运行 generate.sh。你应该看到以下输出：

```
```bash
./generate.sh
org1.example.com
2018-04-06 20:30:19.886 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
2018-04-06 20:30:19.893 CST [common/tools/configtxgen] doOutputBlock -> INFO 002
Generating genesis block
2018-04-06 20:30:19.893 CST [common/tools/configtxgen] doOutputBlock -> INFO 003
Writing genesis block
2018-04-06 20:30:19.916 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
2018-04-06 20:30:19.920 CST [common/tools/configtxgen] doOutputChannelCreateTx ->
INFO 002 Generating new channel configtx
2018-04-06 20:30:19.941 CST [common/tools/configtxgen] doOutputChannelCreateTx ->
INFO 003 Writing new channel tx
2018-04-06 20:30:19.971 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
2018-04-06 20:30:19.978 CST [common/tools/configtxgen] doOutputAnchorPeersUpdate
-> INFO 002 Generating anchor peer update
2018-04-06 20:30:19.978 CST [common/tools/configtxgen] doOutputAnchorPeersUpdate
-> INFO 003 Writing anchor peer update
...`
```

要启动网络，运行 start.sh。你应该看到以下输出：

```
```bash
./start.sh
...
docker-compose -f docker-compose.yml up -d ca.example.com orderer.example.com
peer0.org1.example.com couchdb
Creating couchdb ... done
Creating peer0.org1.example.com ... done
Creating orderer.example.com ...
Creating ca.example.com ...
Creating peer0.org1.example.com ...
wait for Hyperledger Fabric to start
...
Create the channel
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/
hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer
channel create -o orderer.example.com:7050 -c mychannel -f /etc/hyperledger/
configtx/channel.tx
2018-04-06 12:33:25.408 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and
orderer connections initialized
2018-04-06 12:33:25.457 UTC [channelCmd] InitCmdFactory -> INFO 002 Endorser and
orderer connections initialized
2018-04-06 12:33:25.663 UTC [main] main -> INFO 003 Exiting.....
Join peer0.org1.example.com to the channel.`
```

```
docker exec -e "CORE_PEER_LOCALMSPID=Org1MSP" -e "CORE_PEER_MSPCONFIGPATH=/etc/
hyperledger/msp/users/Admin@org1.example.com/msp" peer0.org1.example.com peer
channel join -b mychannel.block
2018-04-06 12:33:25.867 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and
orderer connections initialized
2018-04-06 12:33:25.956 UTC [channelCmd] executeJoin -> INFO 002 Successfully
submitted proposal to join channel
2018-04-06 12:33:25.956 UTC [main] main -> INFO 003 Exiting.....
...

```

要停止它，运行 stop.sh:

```
```bash
./stop.sh
# Shut down the Docker containers that might be currently running.
docker-compose -f docker-compose.yml stop
Stopping peer0.org1.example.com ... done
Stopping couchdb ... done
Stopping orderer.example.com ... done
...

```

运行 teardown.sh，在你的系统上彻底删除网络的所有“犯罪证据”:

```
```bash
./teardown.sh
Removing peer0.org1.example.com ... done
Removing ca.example.com ... done
Removing couchdb ... done
Removing orderer.example.com ... done
Removing network net_basic
...
...

```

basic-network 和 first-network 的对比:

- ❑ basic-network 建立了开发 chaincode 和基本应用程序所需的最小节点数量。它只有一个节点，因此也只有一个单一的组织。first-network 建立了一个拥有多个组织的网络，旨在演示更真实的部署拓扑。
- ❑ 密码材料是不同的，因为 first-network 实际上动态生成新的密码材料，而 basic-network 包括预先生成的密码材料。

## 13.4 Fabcar

本节将会指引你基于 Hyperledger Fabric 网络编写第一个应用程序。

### 13.4.1 编写第一个应用

最基本的区块链网络应用程序需要提供给用户查询账本（包含特定记录）以及更新账本

(添加记录)的功能。

我们的应用程序基于 Javascript, 通过 Node.js SDK 与(账本所在的)网络进行交互。这里将通过三步来指导你编写第一个应用程序。

(1) 启动一个 Hyperledger Fabric 区块链测试网络。在我们的网络中, 需要一些最基本的组件来查询和更新账本。这些组件有 peer 节点、ordering 节点以及证书管理。而 CLI 容器则用来发送一些管理命令。有个简单的脚本将下载并启动这个测试网络。

(2) 学习应用程序中用到的智能合约例子中所用到的参数。智能合约包含的各种功能, 以让我们可以用多种方式和账本进行交互。比如, 我们可以读取整体的数据或者某一部分详尽的数据。

(3) 开发能够查询及更新记录的应用程序。我们提供两个程序例子——一个用于查询账本, 另一个用于更新账本。我们的程序将使用 SDK APIs 来和网络进行交互, 并最终调用这些功能。

完成本例之后, 你应该基本了解了一个使用 Hyperledger Fabric Node.js SDK 并带有智能合约的应用程序, 是如何与 Hyperledger Fabric 网络中的账本进行交互的。

下面, 让我们启动测试网络。

### 13.4.2 下载测试网络 (Getting a Test Network)

访问 Prerequisites 网页并确保你的计算机上已经安装了必需的依赖项。

选择 clone fabric-samples 的工作目录, 运行 clone 命令, 并进入 fabcar 子目录:

```
git clone https://github.com/hyperledger/fabric-samples.git
cd fabric-samples/fabcar
```

fabcar 子目录包含运行示例程序的脚本及程序代码。在该目录运行 ls 命令, 你应该会看到以下内容:

```
chaincode invoke.js network package.json query.js startFabric.sh
```

现在调用 startFabric.sh 来启动网络。

下面命令将会下载并解压 Hyperledger Fabric Docker images, 因此需要几分钟时间来完成。

```
./startFabric.sh
```

为了简洁起见, 我们不会深入了解这个命令的具体细节。下面是关于这个命令的简要说明:

- ❑ 启动 peer 节点、Ordering 节点、证书颁发机构及 CLI 容器。
- ❑ 创建一个通道, 并将 peer 加入该通道。
- ❑ 将智能合约(即链码)安装到 peer 节点的文件系统上, 并在通道上实例化该链码; 实例化会启动链码容器。
- ❑ 调用 initLedger 功能来向通道账本写入 10 辆不同的汽车。



**注意** 这些操作通常由组织或者 peer 的管理员来完成。这个脚本使用 CLI 容器来执行这些命令，但 SDK 中也有相应的支持。具体请参阅 Hyperledger Fabric Node SDK repo 中的示例脚本。

发送 `docker ps` 命令可以显示 `startFabric.sh` 脚本启动的进程。你可以在 Building Your First Network 部分了解有关这些操作的详细信息和机制。在这里，我们专注于应用程序。图 13-1 所示简单描述了一个应用程序与 Hyperledger Fabric 网络交互的过程。

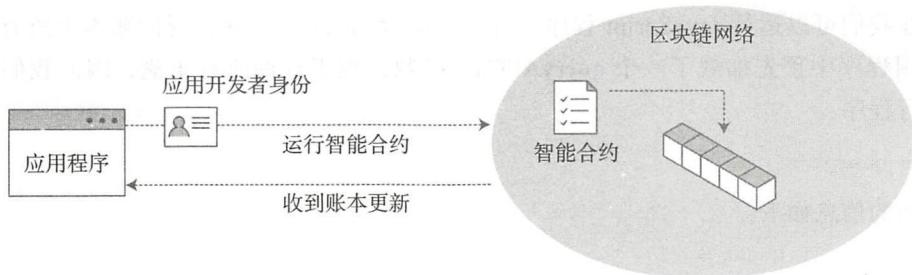


图 13-1 网络交互过程

### 13.4.3 应用程序如何与网络进行交互

应用程序使用 APIs 来调用智能合约（即“链码”）。这些智能合约托管在网络中，并通过名称和版本进行标识。例如，标识为 `dev-peer0.org1.example.com-fabcar-1.0` 的容器，其名称是 `fabcar`，版本号是 `1.0`，运行 `peer` 是 `dev-peer0.org1.example.com`。

API 可通过软件开发工具包（SDK）进行访问。在本节中，我们将使用 Hyperledger Fabric Node SDK，除此以外，Fabric 还提供了 Java SDK 和 CLI 用于开发应用程序。

### 13.4.4 查询账本

查询是指如何从账本中读取数据。你可以查询单个或者多个键的值，如果账本是以类似于 JSON 的数据存储格式写入的，则可以执行更复杂的搜索（如查找包含某些关键字的所有资产），如图 13-2 所示。

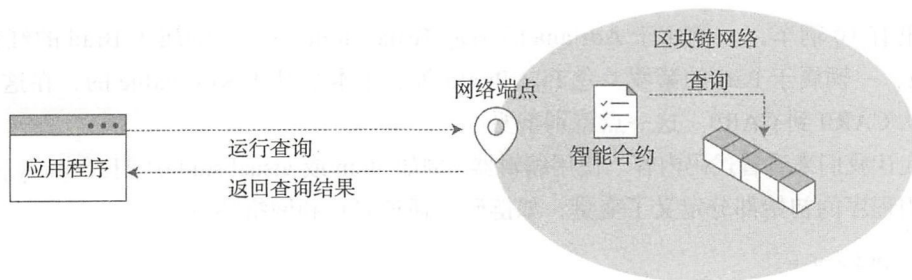


图 13-2 查询过程

正如前面所说，我们的示例网络有一个活跃的链码容器和一个已经包含 10 辆不同汽车的账本。我们还有一些 Javascript 示例代码，我们可以使用 fabcar 目录中的 query.js 来查询账本中关于车的详细信息。

在我们查看该应用程序的工作原理之前，需要安装 SDK 模块。在 fabcar 目录中，运行下面的命令：

```
npm install
```

后续命令也都是在 fabcar 目录中运行的。

现在我们可以运行 JavaScript 程序。首先，运行 query.js 程序，返回账本上所有汽车列表。应用程序中预先加载了一个 queryAllCars 函数，用于查询所有车辆，因此我们可以简单地运行程序：

```
node query.js
```

返回的信息如下：

```
Query result count = 1
Response is [{"Key":"CAR0", "Record":{"colour":"blue","make":"Toyota","model":"Prius","owner":"Tomoko"}},
{"Key":"CAR1", "Record":{"colour":"red","make":"Ford","model":"Mustang","owner":"Brad"}},
{"Key":"CAR2", "Record":{"colour":"green","make":"Hyundai","model":"Tucson","owner":"Jin Soo"}},
{"Key":"CAR3", "Record":{"colour":"yellow","make":"Volkswagen","model":"Passat","owner":"Max"}},
{"Key":"CAR4", "Record":{"colour":"black","make":"Tesla","model":"S","owner":"Adriana"}},
{"Key":"CAR5", "Record":{"colour":"purple","make":"Peugeot","model":"205","owner":"Michel"}},
{"Key":"CAR6", "Record":{"colour":"white","make":"Chery","model":"S22L","owner":"Aarav"}},
{"Key":"CAR7", "Record":{"colour":"violet","make":"Fiat","model":"Punto","owner":"Pari"}},
{"Key":"CAR8", "Record":{"colour":"indigo","make":"Tata","model":"Nano","owner":"Valeria"}},
{"Key":"CAR9", "Record":{"colour":"brown","make":"Holden","model":"Barina","owner":"Shotaro"}}]
```

这里有 10 辆车，一辆属于 Adriana 的黑色 Tesla Model S、一辆属于 Brad 的红色 Ford Mustang、一辆属于 Pari 的紫罗兰色 Fiat Punto 等。账本是基于 Key/Value 的，在这里，关键字是从 CAR0 到 CAR9。这一点特别重要。

现在让我们来看看代码内容。使用编辑器（例如 atom 或 visual studio）打开 query.js 程序。应用程序的初始部分定义了变量，如链码、通道名称和网络端点：

```
var options = {
 wallet_path : path.join(__dirname, './network/creds'),
```

```

user_id: 'PeerAdmin',
channel_id: 'mychannel',
chaincode_id: 'fabcar',
network_url: 'grpc://localhost:7051',

```

这是构建查询的代码块：

```

// queryCar - requires 1 argument, ex: args: ['CAR4'],
// queryAllCars - requires no arguments , ex: args: [''],
const request = {
 chaincodeId: options.chaincode_id,
 txId: transaction_id,
 fcn: 'queryAllCars',
 args: ['']
}

```

我们将 `chaincode_id` 变量赋值为 `fabcar`，这让我们定位到这个特定的链码，然后调用该链码中定义的 `queryAllCars` 函数。

在前面，我们发出 `node query.js` 命令时，会调用特定函数来查询账本。但是，这不是我们能够使用的唯一功能。

要查看其他内容，可转至 `chaincode` 子目录并在编辑器中打开 `fabcar.go`。你会看到 `initLedger`、`queryCar`、`queryAllCars`、`createCar` 和 `changeCarOwner` 等函数。让我们仔细看看 `queryAllCars` 函数是如何与账本进行交互的。

```

func (s *SmartContract) queryAllCars(APIStub shim.ChaincodeStubInterface)
sc.Response {

 startKey := "CAR0"
 endKey := "CAR999"

 resultsIterator, err := APIStub.GetStateByRange(startKey, endKey)

```

该函数调用 `shim` 接口函数 `GetStateByRange` 来返回参数在 `startKey` 和 `endKey` 间的账本数据。这两个键值分别定义为 `CAR0` 和 `CAR999`。因此，我们理论上可以创建 1000 辆汽车（假设 `Keys` 都被正确使用），`queryAllCars` 函数将会显示出每一辆汽车的信息。

图 13-3 所示演示了一个应用程序如何在链码中调用不同功能。

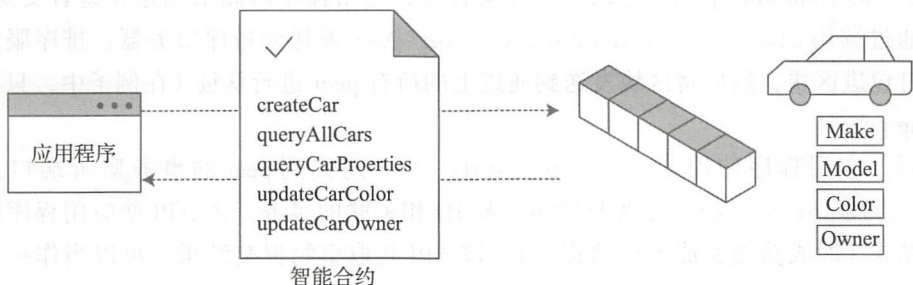


图 13-3 应用程序在链码中的功能



我们可以看到 `queryAllCars` 函数，还有一个称为 `createCar`，这个函数可以让我们更新账本，并最终在链上增加一个新区块。下面先来了解另外一个查询。

现在我们返回 `query.js` 程序并编辑请求构造函数以查询特定的车辆。为达此目的，我们将函数 `queryAllCars` 更改为 `queryCar`，并将特定的“Key”传递给 `args` 参数。在这里，我们使用 `CAR4`。所以我们编辑后的 `query.js` 程序现在应该包含以下内容：

```
const request = {
 chaincodeId: options.chaincode_id,
 txId: transaction_id,
 fcn: 'queryCar',
 args: ['CAR4']
```

保存程序并返回 `fabcar` 目录。现在再次运行程序：

```
node query.js
```

你应该看到以下内容：

```
{ "colour": "black", "make": "Tesla", "model": "S", "owner": "Adriana" }
```

这样，我们就从查询所有车变成了只查询一辆车：Adriana 的黑色 Tesla Model S。使用 `queryCar` 函数，我们可以查询任意关键字（例如 `CAR0`），并获得与该车相对应的制造厂商、型号、颜色和所有者。

现在你应该比较熟悉该链码的基本查询功能以及带参数的查询功能了，下面是时候更新账本了…

### 13.4.5 更新账本

我们已经完成了几个分类账查询并添加了一些代码，现在更新账本。我们可以做很多更新动作，首先让我们为新手创建一辆新车。

账本更新是从生成交易提案的应用程序开始的。就像查询一样，我们将会构造一个请求，用来识别要进行交易的通道 ID、函数以及智能合约。该程序然后调用 `channel.SendTransactionProposalAPI` 将交易建议发送给 `peer(s)` 进行认证。

网络（即 `endorsing peer`）返回一个提案答复，应用程序以此来创建和签署交易请求。该请求通过调用 `channel.sendTransaction` API 发送到排序服务器。排序服务器将把交易打包进区块，然后将区块发送到通道上的所有 `peer` 进行认证（在例子中，只有一个 `endorsing peer`）。

最后，应用程序使用 `eh.setPeerAddr` API 连接到 `peer` 的事务监听端口，并调用 `eh.registerTxEvent` 注册与特定交易 ID 相关联的事务。该 API 使应用程序能获得事务的结果（即成功提交或不成功提交）。该 API 获取事物提交结果，可以当作一个通知机制。



**注意** 这里我们不深入讨论交易细节。有关交易如何最终提交给账本的详细信息，请参阅交易流程文档。

我们初始调用的目标是简单地创建一个新的资产（这里为汽车）。我们有一个独立的用于这些交易的 JavaScript 程序——`invoke.js`。就像查询一样，使用编辑器打开程序并转到构建调用的代码块：

```
var request = {
 targets: targets,
 chaincodeId: options.chaincode_id,
 fcn: '',
 args: [],
 chainId: options.channel_id,
 txId: tx_id
```

我们可以调用函数 `createCar` 或者 `changeCarOwner`。首先我们创建一个红色的 Chevy Volt，并把它归属于 Nick。在账本中我们的 Key 值已经用到了 CAR9，所以这里我们将使用 CAR10。更新代码块如下：

```
var request = {
 targets: targets,
 chaincodeId: options.chaincode_id,
 fcn: 'createCar',
 args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],
 chainId: options.channel_id,
 txId: tx_id
```

保存并运行程序，终端将会输出一些提案响应和交易 ID。但是，我们关心的是：`peer` 发出此事务通知，我们的应用程序通过 `eh.registerTxEventAPI` 接收该通知。现在，如果我们回到 `query.js` 程序并调用带有 CAR10 参数的 `queryCar` 函数，将会看到：

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Nick"}
```

我们来调用最后一个函数 `changeCarOwner`。Nick 很慷慨，他想把他的 Chevy Volt 送给 Barry。所以，我们简单编辑 `invoke.js` 如下：

```
var request = {
 targets: targets,
 chaincodeId: options.chaincode_id,
 fcn: 'changeCarOwner',
 args: ['CAR10', 'Barry'],
 chainId: options.channel_id,
 txId: tx_id
```

再次运行 `node invoke.js`，之后再运行查询程序，我们仍然是使用 CAR10 作为参数查询，将会看到如下结果：

```
Response is {"colour":"Red","make":"Chevy","model":"Volt","owner":"Barry"}
```

## 13.5 Balance transfer

Balance transfer 是一个 Node.js 应用程序示例，用来演示 fabric-client 和 fabric-ca-client Node.js SDK API。

### 13.5.1 预置环境

- ❑ Docker - v1.12 or higher, Docker 版本 1.12 或者更高的版本；
- ❑ Docker Compose - v1.8 or higher, Docker Compose 版本 1.8 或者更高的版本；
- ❑ Git client - needed for clone commands, 有克隆命令的 Git 客户端；
- ❑ Node.js v8.4.0 or higher, Node.js 版本在 8.4.0 或者更高的版本；
- ❑ 下载 docker 镜像。

```
cd fabric-samples/balance-transfer/
```

完成上述设置之后，你将使用以下 Docker 容器配置来配置本地网络：

- ❑ 2 CAs；
- ❑ A SOLO orderer；
- ❑ 4 peers (2 peers per Org)。

### 13.5.2 工件

使用 Hyperledger Fabric 中的 cryptogen 工具生成加密材料，并将其安装到所有对等节点，即 orderer 节点和 CA 容器。关于 cryptogen 工具的更多细节可以在 [http://hyperledger-fabric.readthedocs.io/en/latest/build\\_network.html#crypto-generator](http://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#crypto-generator) 处找到。

一个 orderer 创世区块 (genesis.block) 和通道配置事务 (mychannel.tx) 已使用 Hyperledger Fabric 中的 configtxgen 工具预先生成并放置在工件文件夹中。关于 configtxgen 工具的更多细节可以在 [http://hyperledger-fabric.readthedocs.io/en/latest/build\\_network.html#configuration-transaction-generator](http://hyperledger-fabric.readthedocs.io/en/latest/build_network.html#configuration-transaction-generator) 处找到。

### 13.5.3 运行示例程序

有两种选择可用于运行 balance-transfer 示例，对于这些选择中的每一个，你可以选择使用以 golang 或 node.js 编写的 chaincode 运行。

#### 方法 1

终端窗口 1：使用 docker-compose 启动网络。

```
docker-compose -f artifacts/docker-compose.yaml up
```

终端窗口 2：安装 fabric-client 和 fabric-ca-client 节点模块。



```
npm install
```

在 PORT 4000 上启动节点应用程序：

```
PORT=4000 node app
```

终端窗口 3：从示例 API 请求部分执行 REST API [Sample REST APIs Requests] (<https://github.com/hyperledger/fabric-samples/tree/master/balance-transfer#sample-rest-apis-requests>)。

## 方法 2

终端窗口 1：

```
cd fabric-samples/balance-transfer
./runApp.sh
```

这会在本地计算机上启动所需的网络。安装 fabric-client 和 fabric-ca-client 节点模块，然后，启动 PORT 4000 上的节点应用程序。

终端窗口 2：为了让下面的 shell 脚本正确解析 JSON，你必须安装 jq。请参阅 <https://stedolan.github.io/jq/>。

在终端 1 启动应用程序后，通过执行脚本测试 API——testAPIs.sh：

```
cd fabric-samples/balance-transfer
要使用golang chaincode, 请执行以下命令
./testAPIs.sh -l golang
或者使用node.js chaincode
./testAPIs.sh -l node
```

### 13.5.4 示例——REST APIs 请求

下述请求描述了 Balance transfer 示例，其利用 REST API 暴露了一系列 Fabric 网络操作及查询的接口，方便用户直接通过接口访问。

#### 1. 登录请求

在组织中注册并注册新用户——Org1：

```
curl -s -X POST http://localhost:4000/users -H "content-type: application/x-www-form-urlencoded" -d 'username=Jim&orgName=Org1'
```

输出：

```
{
 "success": true,
 "secret": "RaxhMgevgJcm",
 "message": "Jim enrolled Successfully",
 "token": "<put JSON Web Token here>"
}
```

响应包含了成功/失败状态，一个 enrollment Secret 和一个 JSON Web Token (JWT)，这是后续请求的请求头中必需的字符串。

## 2. 创建通道请求

向 Orderer 节点发送创建通道的请求。

```
curl -s -X POST \
 http://localhost:4000/channels \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
 "channelName": "mychannel",
 "channelConfigPath": "../artifacts/channel/mychannel.tx"
 }'
```

头部 authorization 必须包含从 POST / users 调用返回的 JWT。

## 3. 加入通道请求

将指定的区块链节点加入相应的通道。

```
curl -s -X POST \
 http://localhost:4000/channels/mychannel/peers \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
 "peers": ["peer0.org1.example.com", "peer1.org1.example.com"]
 }'
```

## 4. 安装 chaincode

将链码部署到区块链节点上。

```
curl -s -X POST \
 http://localhost:4000/chaincodes \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
 "peers": ["peer0.org1.example.com", "peer1.org1.example.com"],
 "chaincodeName": "mycc",
 "chaincodePath": "github.com/example_cc/go",
 "chaincodeType": "golang",
 "chaincodeVersion": "v0"
 }'
```



**注意** 当使用 node.js chaincode 时, chaincodeType 必须设置为节点, 并且 chaincodePath 必须设置为 node.js chaincode 的位置。也放在相同目录中。

```
ex:
curl -s -X POST \
 http://localhost:4000/chaincodes \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
```

```

 "peers": ["peer0.org1.example.com", "peer1.org1.example.com"],
 "chaincodeName": "mycc",
 "chaincodePath": "$PWD/artifacts/src/github.com/example_cc/node",
 "chaincodeType": "node",
 "chaincodeVersion": "v0"
 },

```

## 5. 实例化 chaincode

对部署在节点上的链码进行实例化。

```

curl -s -X POST \
 http://localhost:4000/channels/mychannel/chaincodes \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
 "peers": ["peer0.org1.example.com", "peer1.org1.example.com"],
 "chaincodeName": "mycc",
 "chaincodeVersion": "v0",
 "chaincodeType": "golang",
 "args": ["a", "100", "b", "200"]
 }'

```

使用 node.js chaincode 时, chaincodeType 必须设置为节点。

## 6. 调用请求

调用 Fabric 网络中的链码方法。

```

curl -s -X POST \
 http://localhost:4000/channels/mychannel/chaincodes/mycc \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json" \
 -d '{
 "peers": ["peer0.org1.example.com", "peer1.org1.example.com"],
 "fcn": "move",
 "args": ["a", "b", "10"]
 }'

```



**注意** 确保保存响应中的事务 ID, 以便在随后的查询事务中传递此字符串。

## 7. Chaincode 查询

调用链码中相应的查询方法。

```

curl -s -X GET \
 "http://localhost:4000/channels/mychannel/chaincodes/mycc?peer=peer0.org1.
 example.com&fcn=query&args=%5B%22a%22%5D" \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"

```



## 8. 通过 BlockNumber 查询区块

通过 BlockNumber 查询区块信息。

```
curl -s -X GET \
 "http://localhost:4000/channels/mychannel/blocks/1?peer=peer0.org1.example.com" \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"
```

## 9. 通过 TransactionID 查询事务

通过 TransactionID 查询具体交易信息。

```
curl -s -X GET http://localhost:4000/channels/mychannel/transactions/<put
transaction id here>?peer=peer0.org1.example.com \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"
```



**注意** 事务 ID 可以来自任何先前的调用事务，请参阅调用请求的结果，将看起来像 8a95b1794cb17e7772164c3f1292f8410fcfdc1943955a35c9764a21fcd1d1b3。

## 10. 查询 ChainInfo

查询 Fabric 网络中当前区块 Hash、上一个区块 Hah、区块高度。

```
curl -s -X GET \
 "http://localhost:4000/channels/mychannel?peer=peer0.org1.example.com" \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"
```

## 11. 查询已安装的 chaincodes

查询当前 Fabric 网络中已经安装的链码。

```
curl -s -X GET \
 "http://localhost:4000/chaincodes?peer=peer0.org1.example.com&type=installed" \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"
```

## 12. 查询已实例化的 chaincodes

查询当前 Fabric 网络中已经实例化的链码。

```
curl -s -X GET \
 "http://localhost:4000/chaincodes?peer=peer0.org1.example.com&type=instantiated" \
 -H "authorization: Bearer <put JSON Web Token here>" \
 -H "content-type: application/json"
```

## 13. 查询通道

查询 Fabric 中的通道数量。

```
curl -s -X GET \
 "http://localhost:4000/channels?peer=peer0.org1.example.com" \
```

```
-H "authorization: Bearer <put JSON Web Token here>" \
-H "content-type: application/json"
```

## 14. 清理网络

网络仍然在运行。在再次手动启动网络之前，应先用下面的命令清理容器和工件。

```
docker rm -f $(docker ps -aq)
docker rmi -f $(docker images | grep dev | awk '{print $3}')
rm -rf fabric-client-kv-org[1-2]
```

## 15. 网络配置注意事项

可以通过直接编辑 `network-config.yaml` 文件或为可替代的目标网络提供附加的文件来更改配置参数。该应用程序使用可选的环境变量 `TARGET_NETWORK` 来控制要使用的配置文件。例如，如果在 Amazon Web Services EC2 上部署目标网络，则可以添加文件 `network-config-aws.yaml`，并将 `TARGET_NETWORK` 环境变量设置为 `aws`。该应用程序将获取 `network-config-aws.yaml` 文件中的设置。

## 16. IP 地址和端口信息

如果选择通过为节点和 orderer 硬编码 IP 地址以及端口信息来自定义 `docker-compose.yaml` 文件，那么还必须将相同的值添加到 `network-config.yaml` 文件中。`url` 和 `eventUrl` 设置将需要调整以匹配 `docker-compose.yaml` 文件。

```
peer1.org1.example.com:
 url: grpc://x.x.x.x:7056
 eventUrl: grpc://x.x.x.x:7058
```

## 17. 发现 IP 地址

要为其中一个网络实体检索 IP 地址，可发出以下命令：

```
#这将返回peer0的IP地址
docker inspect peer0 | grep IPAddress
```

# 13.6 Hyperledger Fabric CA 示例

Hyperledger Fabric CA 示例演示了以下内容：

- ❑ 如何使用 Hyperledger Fabric CA 客户端和服务端来生成所有加密材料。cryptogen 工具不适用于生产环境，因为它在一个位置生成所有私钥必须将其复制到适当的主机或容器中。本示例演示了如何为 orderers、节点、管理员和端用户生成加密材料，以便私钥永远不会离开生成它们的主机或容器。
- ❑ 如何使用基于属性的访问控制（ABAC）。请参阅 `fabric-samples/chaincode/abac/abac.go` 和注意使用 `github.com/hyperledger/fabric/core/chaincode/lib/cid` 软件包来提取来自调用者身份的属性。仅具有 `abac.init` 属性值的标识 `true` 可以成功调用 `Init` 函数来实例化链码。

### 13.6.1 运行这个示例

1. 运行此示例需要以下镜像: `hyperledger/fabric-ca-orderer`, `hyperledger/fabric-ca-peer`, and `hyperledger/fabric-ca-tools`

#### 1.1.0

运行此示例提供的 `bootstrap.sh` 脚本来下载所需的 `fabric-ca` 示例的镜像。``bash ./bootstrap.sh ==> Pulling fabric ca Image ==> FABRIC CA IMAGE  
==> List out hyperledger docker images ``

#### 1.0.X

这些镜像是 `github.com/hyperledger/fabric-ca` 的 v1.1.0 发行版中的新增功能。为了在 v1.1.0 发行版之前运行此示例, 必须构建这些示例手动镜像如下:

- (1) 拉 `github.com/hyperledger/fabric` 的主分支 `github.com/hyperledger/fabric-ca` 存储库;
- (2) 确保这些存储库位于您的 `GOPATH` 上;
- (3) 运行本示例提供的 `build-images.sh` 脚本。

要运行此示例, 只需运行 `start.sh` 脚本。根据需要, 可以连续多次执行此操作, 因为每次启动前 `start.sh` 脚本都会被清理。你应该看到以下输出:

```
bash ##### 2018-04-06 20:15:01 Deleting existing docker containers ... #####
2018-04-06 20:15:05 Removing chaincode docker images ... ##### 2018-04-06
20:15:05 Cleaning up the data directory from previous run at ./data #####
2018-04-06 20:15:05 Created docker-compose.yml Creating peer2-org2 ... done
Creating run ... done Creating rca-org1 ... Creating rca-org0 ... Creating
ica-org0 ... Creating ica-org1 ... Creating ica-org2 ... Creating setup ...
Creating orderer1-org0 ... Creating peer2-org2 ... Creating peer1-org1 ...
Creating peer1-org2 ... Creating peer2-org1 ... Creating run ...
```

以及成功后会显示:

```
bash ##### 2018-04-06 12:16:49 Congratulations! The tests ran successfully. #####
2018-04-06 12:16:49 See data/logs/run.log for more details
```

要停止由 `start.sh` 脚本启动的容器, 可以运行 `stop.sh` 脚本。

```
bash ./stop.sh ##### 2018-04-06 20:19:51 Stopping docker containers ... ##### ...
2018-04-06 20:20:47 Docker containers have been stopped
```

### 13.6.2 了解这个例子

在 `fabric-samples/fabric-ca/scripts/env.sh` 脚本的顶部有一些变量用于定义此示例的名称和拓扑。可以按照评论中所述修改这些内容的脚本来定制这个示例。默认情况下, 有三个组织: `orderer` 组织是 `org0`, 两个 `peer` 组织是 `org1` 和 `org2`。

`start.sh` 脚本首先构建 `docker-compose.yml` 文件 (通过调用 `makeDocker.sh` 脚本), 然后启动 Docker 容器。



data 目录是所有容器的 volume mount。在实际场景中不需要此 volume mount，此示例使用此 volume mount 的原因如下：

- (1) 所有容器都可以将其日志写入一个公共目录（即 the data/logs 目录），以便于调试；
- (2) 同步容器启动的顺序（例如，ica 容器中的中间 CA 必须等待在 rca 容器中的相应根 CA 将其证书写入 data\* 目录）；
- (3) 访问客户端，通过 TLS 连接所需的引导证书。

在 docker-compose.yml 文件中定义的容器按照以下顺序。

(1) rca（根 CA）容器首先启动，每个组织启动一个。一个 rca 容器为 a 的根 CA 运行 fabric-ca-server 组织。根 CA 证书写入 data 目录，并在中间 CA 通过 TLS 连接到它时使用。

(2) 接下来启动 ica 容器。一个 ica 容器为组织的中间 CA 运行 fabric-ca-server。每个容器都会注册一个对应的根 CA，中间 CA 证书也写入 data 目录。

(3) setup 容器向中间 CA 注册身份、生成创世区块，以及设置该区块链网络所需的其他工件。这由 fabric-samples/fabric-ca/scripts/setup-fabric.sh 脚本完成。注意，管理员身份应使用 abac.init=true:ecert 注册（请参阅此脚本的 registerPeerIdentities 函数），这导致管理员的注册证书（ECert）具有名为“abac.init”的属性值为“真”。进一步注意这个样本使用的 chaincode，其要求将此属性包含在身份证明中来调用它的 Init 函数。请参阅 fabric-samples/chaincode/abac/abac.go\* 上的链码。

有关基于属性的访问控制（ABAC）的更多信息，请参阅 <https://github.com/hyperledger/fabric/tree/release/core/chaincode/lib/cid/README.md>。

(4) orderer 和 peer 容器已启动。这些容器的命名就像它们在 data/logs 目录中的日志文件一样直接。

(5) 启动 run 容器，运行实际的测试用例。测试用例会建一个通道，peer 加入通道，chaincode 被安装和实例化，链码被查询并被调用。大家可参看 fabric-samples/fabric-ca/scripts/run-fabric.sh 脚本的 main 函数获取更多细节。

## 13.7 高性能网络

该网络用于了解如何在每秒处理数千次事务时正确设计链码数据模型，这些事务都会更新账本中的相同资产。一个简单的实现是使用单个键来表示资产的数据，然后链码会在它的事务每次进入时尝试更新该键。但是，当许多事务全部进入时，在 peer 上模拟交易（即创建读取集）并且它已准备好提交到账本时，另一个交易可能已经更新了相同的值。因此，在简单实现中，读取集版本将不再与 orderer 中的版本匹配，并且大量并行事务将失败。为了解决这个问题，经常更新的值被存储为一系列增量值，这些增量值在必须检索时汇总。通过这种方式，不会经常读取和更新单个行，而是会考虑行的集合。

### 13.7.1 用例

链码数据模型设计的主要用例是对特定资产具有频繁增删操作的应用程序。例如，在银行或信用卡账户中，资金会不停流入或流出，而账户中的资金数额是所有这些加和减的汇总结果。一个人的银行账户可能不会经常产生高并发的吞吐量，但是用于存储电子商务平台上客户收取的资金的组织账户可能很快会从全世界接收大量的交易。事实上，这个用例与比特币这样的加密货币的唯一用例类似：用户未使用的交易输出（UTXO）是指自加入区块链已成为其一部分的所有交易。可以采用这种技术的其他使用案例典型代表是 IOT 传感器，这些传感器经常更新其在云中的感知价值。通过采用这种存储数据的方法，组织可以优化其链码以尽可能快地存储和记录交易，并且可以汇总分类账记录在不牺牲交易绩效的情况下，在他们选择的时候将其价值纳入云中。但是，鉴于 Hyperledger Fabric 的状态机设计，需要仔细考虑链码的数据模型设计。

### 13.7.2 如何使用

此示例提供运行高吞吐量应用程序所需的链码和脚本。为了便于使用，尽管做了一些小的修改，它运行在 fabric-samples 内的 first-network 文件夹中 'byfn.sh 启动的同一网络上。在 fabric-samples 内的 first-network 文件夹中 byfn.sh' 下面，提供了构建网络并运行一些调用的说明和一些调用。

#### 1. 构建你的网络

(1) cd 进入 fabric-samples 下的 first-network 文件夹，比如 `cd ~/fabric-samples/first-network`。

(2) 在你喜欢的编辑器中打开 `docker-compose-cli.yaml`。

(3) 在 cli 容器的 volume 部分，编辑第二行，把指向 chaincode 文件夹改成指向 high-throughput 文件夹内的 chaincode 文件夹，例如：

```
../chaincode/:/opt/gopath/src/github.com/hyperledger/fabric/examples/chaincode/go --> ../high-throughput/chaincode/:/opt/gopath/src/github.com/hyperledger/fabric/examples/chaincode/go *
```

再次在 volumes 部分，编辑引用脚本文件夹的第四行，以便它指向 high-throughput 文件夹中的 scripts 文件夹，例如：

```
./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/ --> ../high-throughput/scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
```

(4) 最后，用 # 注释掉 command 部分，例如，`#command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME}; sleep $TIMEOUT'`

(5) 我们现在输入 `./byfn.sh -m up -c mychannel` 来启动我们的网络。

(6) 打开一个新的终端窗口并把 `docker exec -it cli bash` 输入 CLI 容器，从

现在开始，网络上的所有操作都将在此容器内发生。

## 2. 安装和实例化链码

安装和实例化链码的流程如下：

(1) 进入 CLI 容器后，运行 `cd scripts` 来进入 `scripts` 文件夹。

(2) 运行 `setclienv.sh` 来设置环境变量。

(3) 通过运行 `./channel-setup.sh` 来设置通道和锚节点。

(4) 通过运行 `./install-chaincode.sh 1.0` 来安装链码。唯一的参数是每次代表 `chaincode` 版本的数字，若希望安装并升级到新的链式代码版本，则只需在运行该命令时将此值增加 1。比如，`./install-chaincode.sh 2.0`。通过运行 `./instantiate-chaincode.sh 1.0` 来实例化链码。`version` 参数的作用与 `./install-chaincode.sh 1.0` 中的相同，并且应该与上面刚刚安装的 `chaincode` 的版本相匹配。将来，在将链码升级到更新版本时，应该使用 `./upgrade-chaincode.sh 2.0` 来代替 `./instantiate-chaincode.sh 1.0`。

(5) 至此，`chaincode` 已安装并准备好接受调用。

## 3. 调用链码

所有的调用链码都作为 `scripts` 文件夹中的脚本提供，这些在下面详述。

## 4. 更新

更新的格式是：

```
./update-invoke.sh name value operation
```

其中 `name` 是要更新的变量的名称；`value` 是添加到变量的值；而 `operation` 可以是 `+` 或 `-`，具体取决于想要添加到变量的操作类型。在将来，乘法 / 除法操作将被支持（或者将它们自己添加到 `chaincode` 作为练习）。

示例：

```
./update-invoke.sh myvar 100 +
```

## 5. 获取

获取的格式是：

```
./get-invoke.sh name
```

其中 `name` 是要获取的变量的名称。

示例：

```
./get-invoke.sh myvar
```

## 6. 删除

删除的格式是：



```
./delete-invoke.sh name
```

其中 name 是要删除的变量的名称。

示例：

```
./delete-invoke.sh myvar
```

## 7. 修剪

修剪取得为变量生成的所有增量，并将它们全部组合到一行中，删除所有先前的行。这有助于清理当许多更新时被执行的账本。有两种修剪方式：prunefast 和 prunesafe。修剪可快速执行删除并同时进行聚合，因此如果在确保数据完整性的过程中发生错误，Prune 安全地首先执行聚合，备份结果，然后执行删除操作。这样，如果一路上出现错误，数据完整性将得以保持。

修剪的格式是：

```
./[prunesafe|prunefast]-invoke.sh name
```

其中 name 是要修剪的变量名称。

示例：

```
./prunefast-invoke.sh myvar
```

或者

```
./prunesafe-invoke.sh myvar
```

## 8. 测试网络

这里提供了两个脚本来展示一次运行多个并行事务时使用此系统的优点：many-updates.sh 和 many-updates-traditional.sh。

第一个脚本接受与 update-invoke.sh 有相同的参数，但并行重复调用 1000 次。因此，最终值应该是给定的更新值乘以 1000。运行此脚本以确认你的网络正在正常运行。你可以检查 peer 和 orderer 日志并验证是否因不合适的版本而拒绝调用来确认此情况。

第二个脚本 many-updates-traditional.sh 也发送 1000 个事务，但使用传统的存储系统。它会在账本中进行 1000 次排序，每次递增 1（即第一次调用将其设置为 0，最后一次将其设置为 1000）。该期望的是该行的最终值为 999。但是，每次运行此脚本时，最终值都会更改，你会发现 peer 和 orderer 日志中的错误。

还有一个脚本 get-traditional.sh，它只是以传统的方式获得一行的值，没有增量。

示例：./many-updates.sh testvar 100 + --> ./get-invoke.sh 中的最终值应该是 100000，而 ./many-updates-traditional.sh testvar --> ./get-traditional.sh testvar 中的最终值应该是 undefined。

## 部署教程

对官网的案例进行部署，以体现源码和实际操作的呼应关系。

### 14.1 下载部署环境

查看官方文档的 Getting Started 部分。Install Prerequisites 部分讲的是安装案例部署时系统所必须有的软件和环境。这里要做的就是安装一些软件而已，这里不做说明。其余需要下载的有：fabric-samples，包含要部署的 chaincode 例子；运行的 docker 镜像、预编译程序（peer/orderer/configtxgen 等）。

官方文档给出了下载 fabric-samples 的一键式操作：

```
git clone -b master https://github.com/hyperledger/fabric-samples.git
```

如果安装了 Prerequisites 部分的 git，则直接执行此命令即可；若没有，也可手动用浏览器打开 <https://github.com/hyperledger/fabric-samples.git> 这个 github 网址，手动下载即可。下载后会形成一个 fabric-samples 文件夹。

官方文档给出了下载 docker 镜像、预编译程序的一键式的操作：

```
curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0-preview
```

这个命令其实就是定位到 <https://goo.gl/6wtTN5>。这个网址指向的是一个脚本，然后使用 -s 指定版本号为 1.1.0-preview，并使用 bash 执行这个脚本。其实，这个脚本本身是存在于源码目录中的，我们可以手动下载。另外，下文在部署的时候若要顺利自动执行脚本，对某些文件所在路径是有要求的，因此这里下载的 docker 镜像、预编译程序均要放到 fabric-samples 文件夹中（主要是 bin 目录要放到这个文件夹中）。

在 github 上直接搜索 fabric；然后找到 hyperledger/fabric，点击进去，自行选择版本；然后找到 script 目录下的 bootstrap-x.x.x.sh，这个文件其实就是上述网址指向的脚本，x 代表版本数字，这里以 1.0.4 版本为例。该脚本文件各版本间框架基本保持不变，只是有若干变量值不同而已，如版本号 VERSION。打开，可以尝试直接执行（笔者测试时总是出错，比如下载 docker 镜像时提示找不到，因此可以根据这个脚本自己手动下载）。该脚本里面有两个比较重要的变量，一个是版本号 VERSION/CA\_VERSION，一个是适用的系统 ARCH/MARCH。VERSION 是手动指定的，如脚本里的 VERSION=\${1:-1.0.4}，即版本号为 1.0.4。自己可以根据自己的版本进行指定，如改为 1.0.0（但官网上相应版本的资源是否还存在则不确定），也即下文命令中所有 \${VERSION} 处都直接替换成 1.0.4。CA\_VERSION 的值同 VERSION。ARCH 的值的确定方法脚本里也直接给出了，ARCH=\$(echo "\$(uname...)")，自行将 echo "\$(uname...)" 放入自己的系统命令行指向一下就能得到具体的值，比如在笔者这里输出的为 linux-amd64，即下文命令中所有 \${ARCH} 处都直接替换成了 linux-amd64。MARCH 同 ARCH 的操作方法一样，笔者系统的 MARCH 为 x86\_64。变量值确定了，就可以手动执行脚本中的命令了。

脚本主要执行了三项任务，这三项任务均可手动执行：

- ❑ curl https://nexus.hyperledger.org/...tar.gz | tar xz 下载与系统匹配的二进制程序压缩包并解压，如 peer、orderer、configtxgen、configtxlator 和脚本。这里可以替换掉命令中的变量值之后，将 https://nexus.hyperledger.org/...tar.gz 这个网址（其实已经是一个指向 tar.gz 压缩包文件的下载地址了）直接粘贴到下载工具中自行下载，然后自己解压。这里形成了一个 bin 目录，把这个 bin 目录放入 fabric-samples 文件夹下即可。这些都是编译好的现成的程序，而关于如何手动编译这些程序，下文会详述。
- ❑ dockerFabricPull \${FABRIC\_TAG}，下载所有与 hyperledger/fabric 有关的 docker 镜像。在这个函数中，循环执行了 docker pull hyperledger/fabric-\$IMAGES:\$FABRIC\_TAG 和 docker tag hyperledger/fabric...，即先下载镜像，后将镜像改为指定的标签，改标签这一步可做可不做。\$IMAGES 变量分别是代码 for IMAGES in peer orderer couchdb ccenv javaenv kafka zookeeper tools; 处指定的这些值，也即要循环下载 peer、orderer、couchdb 等镜像，\$FABRIC\_TAG 则是 \$MARCH-\$VERSION 的组合，在笔者的系统中为 x86\_64-1.0.4，也即手动逐条执行 docker pull hyperledger/fabric-peer:x86\_64-1.0.4，docker pull hyperledger/fabric-orderer:x86\_64-1.0.4...即可。这里需要说一句，笔者遇到过这样的情况：同样的 docker pull 命令，有时会出现“没有指定的镜像”的错误，各位若遇到类似错误，可以多试几次。
- ❑ dockerCaPull \${CA\_TAG}，下载 CA 镜像，原理与上一步相同。

如此，即可手动下载官网案例部署所需的程序、脚本、镜像。需要强调的是，因为 Fabric 项目正在开发过程中，因此版本号也在不断变，有些版本对应的资源可能在调整中无法下载，或可能已经删除等，因此版本号的值尽量选择比较新的，或选择 Fabric 在



github 中呈现的版本号。

14.2 编译 peer、orderer、configtxgen 等程序

本节直接介绍项目的 Makefile 构成，至于具体的编译过程，若读懂 Makefile，则一切都不再是问题。这里假设各位对 Makefile 有最简单的了解，比如知道 Makefile 的基本书写规则以及 make 依据判定是否重新编译的规则。

Makefile 相关目录文件如表 14-1 所示。

表 14-1 Makefile 相关目录文件

文 件	说 明
Makefile	主编译文件，make 执行的主体
docker-env.mk	Makefile 直接包含的一个子 Makefile 文件，用于定义编译期间使用的关于 Docker 的一系列数据
scripts/	目录，包含 Makefile 编译时调用的一些 shell 脚本
images/	目录，包含 Makefile 编译时使用的一些用于生成 Dockerfile 文件的 Dockerfile.in 文件，Dockerfile 文件则是用于生成各个 docker 镜像的模板文件
gotools/	目录，包含一个子 Makefile 文件，该子 Makefile 文件用于安装 Makefile 编译时使用的各种第三方 go 工具，如 ginkgo、gocov、govendor 等

依然遵循 target ... : prerequisites ... 这个基本规则。Makefile 文件开头的注释中，罗列了所有的 target，下文以编译 peer 节点为例，即 make peer 命令，其余 target 的形式与 peer 如出一辙。在具体编译 peer 的过程中，target 的依赖项可以是另一个子 target，而子 target 又可以有子 target，因此整个过程是一个树形发散的过程。因此为了标记清楚编译执行的顺序路径，这里用 xxx.xxx.xxx 的形式标记每个 target，如 1 表示顶层的 target，1.1 就表示 1 下面的第一个子 target，1.2 就表示 1 下面的第二个子 target，1.1.1 则表示 1.1 下面的第一个子 target，依此类推。

先整体说一下 Makefile。以 include docker-env.mk 为线，可将 Makefile 分为上下两部分：

(1) 上面部分，无论是 =、?=、+= 还是 :=，均是对一个下文会使用的变量的赋值操作而已，它们之间的差别大家可自行搜索。include docker-env.mk 即在 Makefile 中包含了 docker-env.mk 文件，从名字上即可看出该文件定义了一系列运行 docker 命令时的环境变量数据，其中有对 .dummy-xxx 文件的设计初衷的解释，大概意思就是基于 docker 容器编译输出的文件并不算是标准的系统文件（主要在时间上会和实际系统中参与编译的文件不符），因此需要对容器输出的文件创建对应的 .dummy-xxx 文件，让 .dummy-xxx 文件真正参与 make 的编译。另外，在 Makefile 中到处使用的 \$DUMMY 变量也在 docker-env.mk 中定义。

(2) 下面部分，遵循 target ... : prerequisites ...，是 makefile 的执行主体。

## 14.3 部署

切换到 `fabric-samples/first-network` 目录下，执行脚本操作（由于可能存在系统权限问题，最好以 `sudo` 执行所有的脚本、命令）：

（1）执行 `byfn.sh` 脚本，生成启动、运行所需的配置或数据，如 `channel.tx`、`genesisblock` 等。`sudo ./first-network/byfn.sh -m generate`，这里可以加入一些 flag：① `-c channelDIY`，指定 `channel` 的 ID 为 `channelDIY`，若不指定默认为 `mychannel`。② `-t 10000`，指定 CLI 容器自动退出的时间，若不指定默认为 20000 秒。③ `-s couchdb`，指定使用的数据库为 `couchdb`，若不指定则默认为 `goleveldb`。其他的参数可参看 `byfn.sh` 中的 `printHelp()` 函数中的打印（即执行 `./first-network/byfn.sh --help` 显示的内容）。

（2）执行 `byfn.sh` 脚本，启动运行环境。`sudo ./byfn.sh -m up`，第一次执行这一步的时候，需要确保自己的电脑是联网状态，因为启动过程中仍然需要下载一些东西，如 `fabric-ca`、`fabric-baseos` 镜像。但是之后再次执行时，由于该下载的都已经下载过了，因此电脑是否联网已无所谓。

首先关注 `byfn` 这个名字，其是 `build your first network` 的首字母缩写，寓意很清晰。`byfn.sh` 脚本具体都做了哪些工作，分析这个脚本即可，然后我们也可以顺着这个脚本手工执行。`byfn.sh` 开篇处有两个 `export`，指定了脚本执行过程中寻找程序文件的路径（`./` 和 `../bin`），也指定了 `Fabric` 的配置路径 `FABRIC_CFG_PATH` 的值（`./`）。文件分为 3 部分：第 1 部分是定义各种操作的函数体，如 `networkUp`、`networkDown`、`generateCerts` 等；第 2 部分从 `while getopts "h m:c:t:d:f:s:" opt; do` 循环开始，分析执行该脚本时携带的各种参数，如上文第 1 点分析的那些参数；第 3 部分处于脚本最底部，从 `if [ "${MODE}" == "up" ]; then` 开始，根据第 2 步所确定的 `MODE` 值，在每个 `if` 分支中执行 `up`、`down` 还是 `generate` 操作，这里只看与 `generate` 操作对应的 `generateCerts`、`replacePrivateKey`、`generateChannelArtifacts` 三个函数和与 `up` 操作对应的 `networkUp` 函数。

## 14.4 Crypto Generator

`generateCerts` 函数主要使用了 `cryptogen` 工具。在检查 `cryptogen` 工具和 `crypto-config` 是否存在后（在脚本开始时 `export` 指定的路径中寻找），执行了 `cryptogen generate --config=./crypto-config.yaml`，即 `bin/cryptogen` 工具根据 `first-network/crypto-config.yaml` 配置文件生成 `fabric` 网络拓扑结构和各个组织的根证书。这个网络拓扑结构，指的是你这个区域链初始化时有哪些组织，组织的域名是什么，组织下有多少实体成员。这个工具相应生成每个组织及组织下实体所使用的身份证书、TLS 证书。最终在 `first-network` 下生成了一个 `crypto-config` 文件夹。



### 14.4.1 crypto-config.yaml

以组织为单位，如 OrdererOrgs、PeerOrgs。在每个组织下可配置一个个实体，这个实体等同网络中的一个入口，或者说一个节点，如 OrdererOrgs 下配置的多个 orderer 节点：

- ❑ Name: 组织名。
- ❑ Domain: 域名。
- ❑ Specs: 详细说明，更细致地对每个实体名字进行配置。
- ❑ Hostname: 主机名，必填（以 Hostname.Domain 格式形成实体名）。
- ❑ CommonName: 主机通用名，可选。若定义，则将覆盖上面 Hostname.Domain 格式的实体名。
  - Template: 模板，批量地对多个实体的名字进行配置。
  - Count: 实体的数量，如 5。
  - Start: 从哪个数开始。若不定义，则默认从 0 开始。
- ❑ Hostname: 主机名，若这里定义了，Count 和 Start 的定义将无效而只配置此名。
- ❑ User: 普通用户，该配置区域只针对 PeerOrgs 组织。其中，Count 用于配置普通用户的数量，即 Specs 配置实体数 + Template 配置实体数 - Admin 个数。

Specs 下可定义多个实体，每个实体都可以包含 Hostname、CommonName 一对字符串，其中 CommonName 是可选的。若未定义 CommonName，则 Hostname.Domain 将是该实体证书的 CN 值，若定义了 CommonName，则 CommonName 将是该实体证书的 CN 值。Template 中会从 Start 开始，定义 Count 个 ordererN.Domain 格式名字的实体， $Start \leq N < Count + Start$ 。Template 与 Specs 不互斥，可同时定义，所定义的实体均会被配置，但是注意两个区域所配置形成的实体名不要重复，否则会形成覆盖。User 中，Admin 实体数不允许配置，默认为 1 个，普通用户的数量也可不遵循上述公式。

### 14.4.2 crypto-config 文件夹

crypto-config 文件就是对之后运行起来的整个区域链的网络拓扑结构很好的描述：crypto-config 有两个文件夹，表示有两类组织，orderer 组织 ordererOrganizations 和 peer 组织 peerOrganizations。以 peerOrganizations 为例，又包含两个文件夹，表示该类组织包含两个组织，org1.example.com 和 org2.example.com。以 org1.example.com 为例，其包含组织根证书文件夹 ca、组织中所有的 MSP 身份文件夹 msp、组织中所有的 TLS 网络证书文件夹 tlscac、组织中所有成员角色的 MSP 身份文件夹 users、组织包含的实体节点文件夹 peers。在 peers 中，又包含两个文件夹，分别代表两个实体节点，即 peer0.org1.example.com 和 peer1.org1.example.com。以 peer0.org1.example.com 为例，包含节点 MSP 身份文件夹、节点网络证书文件夹 tls。如此，一级一级地形成管理、对应关系，比如 peer 组织的 org1 中，实体成员 peer0 与 peer1 的 MSP 身份文件夹中的管理员证书应该一样，且应该等





于 org1 的 msp/admincerts 下的证书。再比如 peer0 的根证书是 msp/cacerts/ca.org1.example.com-cert.pem, 从名字上即可看出 peer0 使用的是 org1 组织的证书, 如此的话, 与 peer0 同级的 peer1 使用的也应该是所属组织 org1 的证书。再比如, peer0 属于 org1 组织, 则 peer0 的自有的签名证书 msp/signcerts/peer0.org1.example.com-cert.pem 理应由 org1 认证的证书, 即是由 org1.example.com/ca/ca.org1.example.com-cert.pem 认证签发的。对于此点我们可以自己写一个小程序验证一下, 即分别读取两个证书, 然后调用 peer0 证书的 Verify() 函数验证, 源码参看 verify\_cert.go。你若是在 Ubuntu 环境下, 可以找到该证书, 直接点开 peer0 的证书, Ubuntu 中有解析证书的软件, 在 Issuer Name (发行者) 栏中可以清楚看到组织 O 为 org1.example.com, CN (通用名称) 为 ca.org1.example.com, 即 org1 的证书。事实证明 peer0 的证书确实是 org1 签发的。另外, 组织、实体节点等的命名均是 xxx.xxx.xxx 格式, 这也隐含了前面的在后面的范围之内的意思。

replacePrivateKey 函数所做的主要是在 generateCerts 生成各个实体的证书和私匙后, 替换部署的 fabric-ca0、fabric-ca1 容器所使用的配置文件中指定的与容器所使用的证书对应的私匙文件名。这一点也很好理解: fabric-ca 容器是用于给新加入某一组织的 peer 节点签发证书的, 因此 ca 容器首先得持有组织的根证书和根证书的私匙。根证书的文件名根据 cryptogen 程序可预先得知, 但是对应生成的私匙 (\_sk) 文件名则事先是不知道的, 因此在 docker-compose-e2e-template.yaml (带有 template 字样, 表明是一个样板文件) 中预先是用 CA1\_PRIVATE\_KEY、CA2\_PRIVATE\_KEY 这样的字段临时代替了私匙文件名, 也因此需要在生成 org1 和 org2 两个组织的证书的私匙后, 才能用私匙文件名替换掉这两个字段。具体的操作是: 先复制了一份 docker-compose-e2e-template.yaml 模板文件为 docker-compose-e2e.yaml, 在相应目录中执行 PRIV\_KEY=\$(ls \_sk) 获取私匙文件名, 然后使用 sed \$OPTS "s/... 命令将 docker-compose-e2e.yaml 中的 CA1\_PRIVATE\_KEY、CA2\_PRIVATE\_KEY 字段分别替换成与 org1、org2 组织的根证书对应的私匙文件名。

## 14.5 Configuration Transaction Generator

generateChannelArtifacts 函数主要使用了 configtxgen 工具。函数所做的事情和 configtxgen 工具的介绍可以参看脚本中该函数上面的注释。在此简单说一下, configtxgen 使用 configtx.yaml 配置生成部署的网络实体所需的手工艺品, 这些生成的文件均放入了 channel-artifacts 文件夹内:

- ❑ genesis.block: 这是 orderer 服务启动必备的。特别注意, 每个组织的根证书都包含在 genesis.block 中。
- ❑ channel.tx: fabric 的 channel 配置。该配置用于在 orderer 服务启动创建 channel 时配置 channel。这里就已经形成了 channel 的读写策略 (即哪些实体可以读, 哪些实体可以写, 下同)。



❑ `Org1MSPanchors.tx` 和 `Org2MSPanchors.tx`：用于指定在 channel 中，当前 peer 组织中都有哪些 peer 节点。这里就已经形成了组织的读写策略。

`configtx.yaml` 包含部署网络实体的定义，主要分为三部分：Profiles，直接供 `configtxgen` 使用的配置部分，可以通过 `-profile` 命令行参数指定配置这里的哪一项；Organizations，定义了三个成员（组织），orderer 组织 `OrdererOrg`、两个 peer 组织 `org1` 和 `org2`（每个组织管理维护两个 peer 节点），供 Profiles 使用；其余的 `Orderer` 和 `Application`，包含了一些具体的定义，供 Profiles 和 Organizations 两部分使用。这里应注意，每个组织都配置了 `MSPDir` 项，即每个组织的 MSP 所在的路径，这正是为了 `configtxgen` 能够找到每个组织的根证书，从而可以对实体身份或实体交易的数字签名进行验证。

`configtx.yaml` 配置文件（的写法）与 `configtxgen` 的实现紧密相关，可参看源码 `common/configtx/tool/configtxgen/main.go`：

（1）`configtxgen` 默认就是读取 `configtx.yaml` 作为配置文件，在 `main` 中，`config := genesisconfig.Load(profile)` 默认就读取了当前文件夹或 `FABRIC_CFG_PATH` 或 `fabric/sample-config/`（参看 `byfn.sh` 开始处的两个 `export`，通过三个路径依次搜索指定的配置文件，先搜索到即停止）下的名为 `configtx` 的文件（也就是 `configtx.yaml`）的 `profile` 配置项（`profile` 由 `-profile` 参数指定）。

（2）在 `main` 中，所有的 `flag.StringVar(...)` 即为 `configtxgen` 可以解析的命令行参数，其中就有 `outputBlock`、`outputCreateChannelTx`、`outputAnchorPeersUpdate`。比如，执行 `configtxgen` 时给的 `-outputBlock` 参数，则在 `main` 的中会进入 `if outputBlock != ""` 分支，从而执行 `doOutputBlock(config, channelID, outputBlock)` 来生成 `genesisblock`。另外还有两个用于检查的参数——`inspectBlock`、`inspectChannelCreateTx`，分别用于检查生成的 `genesis.block` 和 `channel.tx` 文件是否符合要求。

（3）对于生成的 `genesisblock`、`channel.tx` 都包含什么数据，建议执行同目录下的 `main_test.go` 中的 `TestBlockFlags`、`TestConfigTxFlags` 两个测试函数。在测试过程中读取的是 `sample-config` 下的 `configtx.yaml` 中的配置，并会将生成的 `genesis.block` 和 `channel.tx` 以 JSON 格式打印出来。

重看 `generateChannelArtifacts` 函数，在检查 `configtxgen` 工具是否存在后，开始执行任务：

（1）`configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block`，生成 `genesisblock`。

❑ `-profile` 指定使用 `configtx.yaml` 中的 `TwoOrgsOrdererGenesis` 项配置去生成 `genesisblock`。

❑ `-outputBlock` 则指定了生成的 `genesisblock` 放到哪里和所用的名字。

（2）`configtxgen -profile TwoOrgsChannel -outputCreateChannelTx...`，使用 `configtx.yaml` 中的 `TwoOrgsChannel` 生成 `channel.tx`。

（3）执行了两个 `configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate...` 命





令, 使用 configtx.yaml 中的 TwoOrgsChannel 配置生成 org1、org2 的配置文件 Org1MS-Panchors.tx 和 Org2MSPanchors.tx。

## 14.6 networkUp- 启动 Fabric 网络

networkUp 函数真正启动了部署网络。函数所做的很简单(前提是熟悉 docker 相关操作), 通过 `if [ ! -d "crypto-config" ]; then` 检查 crypto-config 文件夹是否存在, 也即变相检查上文讲述过的 generateCerts、replacePrivateKey、generateChannelArtifacts 三个函数的准备工作是否已经做完。然后根据是否使用 couchdb 数据库, 执行对应 if 分支中的 ... docker-compose -f ... 命令, 其实就是除了 -f \$COMPOSE\_FILE 外是否多添加一个 -f \$COMPOSE\_FILE\_COUCH 参数, 即多指定一个 docker-compose-couch.yaml 文件以启动 couchdb 容器。-f \$COMPOSE\_FILE 默认指定的是 docker-compose-cli.yaml, 也可在 byfn.sh 的命令行参数中以 -f 指定。docker-compose -f 命令所做的就是根据指定的 docker-compose-cli.yaml 文件自动部署文件中定义的容器。

## 14.7 运行容器 + 区域链操作

关于 docker-compose-cli.yaml, 在 byfn 网络中的 services 下共设置了 6 个服务: 1 个 orderer 容器, 4 个 peer 容器, 1 个 cli 容器。其中 orderer 和 peer 容器都定义在 base/docker-compose-base.yaml 中, cli 则在当前文件中定义。容器的每个字段的定义都没有冗余的地方, 每个字段都很重要, 具体在用到的时候会介绍。docker-compose-couch.yaml 中, 除了为每个 peer 节点容器都对应启动一个 couchdb 容器外, 还为每个 peer 容器增加了一些环境变量。一旦 networkUp 函数执行成功, 所有的容器、容器的设置都会被应用和启动。

在启动各个容器时, docker-compose-cli.yaml 中每个服务下的 command 字段都给定了默认执行的命令。orderer 容器默认执行的是 orderer (也即 orderer start 命令), 4 个 peer 容器各自执行的是 peer node start, 最后启动的 cli 容器, 在 working\_dir 目录下默认执行了 /bin/bash -c './scripts/script.sh \${CHANNEL\_NAME} \${DELAY}; sleep \$TIMEOUT', 即 scripts 下的 script.sh 脚本。但是, 当 script.sh 脚本运行结束之后, cli 容器即会自动停止并退出。script.sh 的执行就是各位在终端上看到的以 START 开始并以 END 结束的过程。这个过程主要执行了几种区域链操作, 如创建 channel、加入 channel、升级组织配置、安装部署 chaincode、调用 chaincode, 算是 cli 先变相测试一下这个部署的区域链网络的基本操作, 同时把 peer1.org1 和 peer1.org2 两个节点加入各自组织(回忆一下, 生成 genesis.block、channel.tx 所使用的 configtx.yaml 中, org1, org2 组织下均只包含了一个 peer0 节点)的状态留在了还在运行的网络里面(一个 orderer 节点容器, 4 个 peer 节点容器仍在运行)。

由此我们可以知道, 当你在终端上看到 START 时, 示例中的 6 个容器已经启动, 整个





实例的网络实际上已经部署完毕，再执行 `script.sh` 脚本只是为了在这个区域链网络中测试各种操作。如果想手动自己操作，在 `docker-compose-cli.yaml` 中将 `cli` 服务中的 `command` 字段注释掉即可，甚至不启动 `cli` 容器也可以，转而在各个节点的容器中执行操作。这里需要注意的是，`script.sh` 脚本是在 `cli` 容器内运行的，也就是说脚本所使用的环境变量、路径等元素的值，均为 `cli` 容器中的值。而命令中 `script.sh` 后边的三个变量 `${CHANNEL_NAME}`、`${DELAY}`、`$TIMEOUT` 则来自于 `byfn.sh` 中 `networkUp` 内执行的 `... docker-compose -f ...` 命令。

由于 `script.sh` 脚本进行了区域链网络的基本操作，因此我们可以细细分析，从中学习，若产生问题，可以适当地修改脚本，验证猜想从而解除疑问。

再回头分析一下 `docker-compose-cli.yaml` 中对 `cli` 容器的设置：`container_name` 指定了容器名字为 `cli`；`image` 指定了 `cli` 以下载的 `hyperledger/fabric-tools` 为镜像；`tty` 也设置为 `true`（即开启了控制台）；在 `environment` 下设置了 `cli` 启动时所设置的环境变量；`working_dir` 指定了 `cli` 的基准工作目录；`command` 指定了 `cli` 启动时运行的命令；`volumes` 指定了宿主主机与 `cli` 之间挂接的目录；`depends_on` 指定了 `cli` 所依赖的容器（即必须在这些容器正确启动之后再启动 `cli`，即 `cli` 是最后启动的）；`networks` 指定了 `cli` 所使用的网络。

在 `script.sh` 中，从 `echo "Creating channel..."` 处开始执行命令，均为调用脚本上面已定义好的函数，这些函数有：

- ❑ `setGlobals()`：公用函数，根据所给的参数来配置针对某一 `peer` 节点的环境变量，以正确执行其他操作。这里预设的是给一个参数，0 对应 `peer0.org1`，1 对应 `peer1.org1`，2 对应 `peer0.org2`，3 对应 `peer1.org2`。比如调用 `setGlobals 0`，表示针对 `peer0.org1` 设置 `cli` 容器的 `CORE_PEER_LOCALMSPID`（`peer` 节点所在组织的路径）、`CORE_PEER_TLS_ROOTCERT_FILE`（`peer` 节点的 `TLS` 证书所在路径）、`CORE_PEER_MSPCONFIGPATH`（`peer` 节点的 `MSP` 配置所在路径）、`CORE_PEER_ADDRESS`（`peer` 节点的容器通信地址）等环境变量。调用者需要根据这些环境变量的值继续执行其他操作。
- ❑ `createChannel()`：`setGlobals 0` 设置了关于 `peer0.org1` 的环境变量，因为 `CORE_PEER_TLS_ENABLED` 的值默认是 `true`（启用 `TLS` 连接），因此执行了 `peer channel create...-o...-c...-f...-tls...` 命令，从而建立了一个与 `orderer` 节点相连的 `Application Channel`。`channel` 的名字由 `$CHANNEL_NAME` 指定，该变量来自于 `cli` 容器执行 `script.sh` 时所给的第 1 个参数，这里第 1 个参数来自执行 `byfn.sh` 时所给的 `-c` 参数值。若不指定，默认值是 `mychannel`，这里假设使用的是默认值。创建 `mychannel` 后，会在 `cli` 容器内的 `working_dir` 下生成一个 `mychannel.block`，供之后要加入的 `mychannel` 的节点使用。`mychannel.block` 中包含了 `channel.tx` 中的配置信息。
- ❑ `joinChannel()`：循环 0~3 这四个值，分别代表 4 个 `peer` 节点，在循环中依次调用“`setGlobals $ch, joinWithRetry $ch`”将每个 `peer` 节点加入上一步所创建的

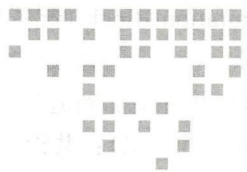


mychannel 中。joinWithRetry 中具体执行的命令为 `peer channel join -b $CHANNEL_NAME.block`，其中 `$CHANNEL_NAME.block` 即是 `createChannel` 生成的 mychannel.block。

- ❑ `updateAnchorPeers()`：使用之前生成的 `Org1MSPanchors.tx`、`Org2MSPanchors.tx` 升级 mychannel 中现有组织的信息（参看 `configtx.yaml` 中的生成升级包所使用的 `TwoOrgsChannel` 下包含的组织 `org1`、`org2` 的信息，如 `MSPDir` 等）。使用的命令为 `peer channel update -o ... -c ... -f...`。`Org1MSPanchors.tx`、`Org2MSPanchors.tx` 就算是针对组织的升级包，可以升级组织的配置信息，又由于生成的 `Org1MSPanchors.tx`、`Org2MSPanchors.tx` 与创建 mychannel 的 `channel.tx` 所使用的组织信息一致，所以这里的升级实际上不会对现有部署的网络产生任何影响，这里只是为了测试 `peer channel update` 这项功能而已（也即在 `script.sh` 中将 `updateAnchorPeers 0` 和 `updateAnchorPeers 2` 注释掉也没有关系）。
- ❑ `installChaincode()`：使用指定的节点安装一个 chaincode，经过 cli 的路径映射，安装的 chaincode 实际上下载了 `fabric-sample/chaincode/chaincode_example02`。具体执行的命令为 `peer chaincode install -n mycc -v 1.0 ...`，分别在 `peer0.org1` 和 `peer0.org2` 两个节点上安装。
- ❑ `instantiateChaincode()`：从指定节点部署安装的 `chaincode_example02`，这里是从 `peer0.org2` 节点部署。具体执行的命令是 `peer chaincode instantiate -o ... --tls true --cafile... -C...-n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"`。其中，`-c` 指定了初始化的参数，初始化了账户 a 的余额为 100，账户 b 的余额为 200；`-P` 指定了 chaincode 的背书策略，因为是 OR（即，或），所以 `org1` 或 `org2` 下的任一成员背书交易即可，这个意思就是 `org1` 或 `org2` 组织下的所有 `peer` 节点都能执行这个部署的 chaincode。
- ❑ `chaincodeQuery()`：以第一个参数 `$1` 指定节点身份执行查询，对比查询的结果，即 a 账户的余额是否等于第二个参数 `$2`。在 `$TIMEOUT` 内每隔 `$DELAY` 秒会尝试查询一次，每次执行的命令都是 `peer chaincode query -C ... -n mycc -c '{"Args":["query","a"]}'`。
- ❑ `chaincodeInvoke()`：从指定节点执行 `chaincode_example02`。具体执行的命令是 `peer chaincode invoke -o ... --tls true --cafile ... -C ... -n mycc -c '{"Args":["invoke","a","b","10"]}'`，所做的就是将 a 账户的 10 元转账给 b 账户（`chaincode_example02` 源码中所实现的）。







## 专业术语

(1) Anchor Peer (锚节点): 锚节点是通道中能被所有对等节点探测, 并能与之进行通信的一种对等节点。通道中的每个成员都有一个 (或多个, 以防单点故障) 锚节点, 允许属于不同成员身份的节点来发现通道中存在的其他节点。

(2) Block (区块): 在一个通道上, 区块是一组有序交易的集合。区块往往通过密码学手段 (Hash 值) 连接到前导区块。

(3) Chain (链): chain 就是 block 之间以 Hash 连接为结构的交易日志。peer 从 order service 接收交易 block, 并根据背书策略和并发冲突标记 block 上的交易是否有效, 然后将该 block 追加到 peer 文件系统中的 hash chain 上。

(4) Chaincode (链码): 链码是一个运行在账本上的软件, 它可以对资产进行编码, 其中的交易指令 (或者叫业务逻辑) 也可以用来修改资产。

(5) Channel (通道): 通道是构建在 Fabric 网络上的私有区块链, 实现了数据的隔离和保密。通道特定的账本在通道中是与所有对等节点共享的, 并且交易方必须通过该通道的正确验证才能与账本进行交互。通道是由一个配置块来定义的。

(6) Commitment (提交): 一个通道中的每个对等节点都会验证交易的有序区块, 然后将区块提交 (写或追加) 至该通道上账本的各个副本。对等节点也会标记每个区块中的每笔交易的状态是有效或者无效。

(7) Concurrency Control Version Check (并发控制版本检查, CCVC): CCVC 是保持通道中各对等节点间状态同步的一种方法。对等节点并行的执行交易, 在交易提交至账本之前, 对等节点会检查交易在执行期间读到的数据是否被修改。如果读取的数据在执行和提交之间被改变, 就会引发 CCVC 冲突, 该交易就会在账本中被标记为无效, 而且值不会更新到状态数据库库中。





(8) Configuration Block (配置区块): 包含为系统链(排序服务), 或通道定义成员和策略的配置数据。对某个通道或整个网络的配置修改(比如, 成员离开或加入)都将导致生成一个新的配置区块并追加到适当的链上。这个配置区块会包含创始区块的内容加上增量。

(9) Consensus (共识): 共识是贯穿整个交易流程的广义术语, 其用于产生一个对于排序的同意书和确认构成区块的交易集的正确性。

(10) Current State (当前状态): ledger 的 current state 表示其 chain 交易 log 中所有 key 的最新值。peer 会将处理过的 block 中的与每个交易对应的修改 value 提交到 ledger 的 current state, 由于 current state 表示 channel 所知的所有最新的 k-v, 所以 current state 也被称为 World State。Chaincode 执行交易 proposal 就是针对的 current state。

(11) Dynamic Membership (动态成员): Fabric 支持动态添加、移除 members、peers 和 ordering 服务节点, 而不会影响整个网络的操作性。当业务关系调整或因各种原因需添加、移除实体时, Dynamic Membership 至关重要。

(12) Endorsement (背书): Endorsement 是指一个 peer 执行一个交易并返回 Yes 或 No 给生成交易 proposal 的 client App 的过程。chaincode 具有相应的 endorsement policies, 其中指定了 endorsing peer。

(13) Endorsement policy (背书策略): Endorsement policy 定义了依赖于特定 chaincode 执行交易的 channel 上的 peer 和响应结果 (endorsements) 的必要组合条件 (即返回 Yes 或 No 的条件)。Endorsement policy 可指定对于某一 chaincode 对交易背书的最小背书节点数或者最小背书节点百分比。背书策略由背书节点基于应用程序和对抵御不良行为的期望水平来组织管理。在进行 install 和 instantiate Chaincode (deploy tx) 时需要指定背书策略。

(14) Fabric-ca: Fabric-ca 是默认的证书管理组件, 它向网络成员及其用户颁发基于 PKI 的证书。CA 为每个成员颁发一个根证书 (rootCert), 为每个授权用户颁发一个注册证书 (eCert), 为每个注册证书颁发大量交易证书 (tCerts)。

(15) Genesis Block (初始区块): Genesis Block 是初始化区块链网络或 channel 的配置区块, 也是链上的第一个区块。

(16) Gossip Protocol (Gossip 协议): Gossip 数据传输协议有三项功能: 管理 peer discovery 和 channel 成员; 在 channel 上所有 peer 间广播账本数据; 在 channel 上所有 peer 间同步账本数据。

(17) Initialize (初始化): 一个初始化 chaincode 程序的方法。

(18) Install (安装): 将 chaincode 放到 peer 的文件系统的过程 (即将 Chaincode-DeploymentSpec 信息存到 chaincodeInstallPath-chaincodeName.chainVersion 文件中)。

(19) Instantiate (实例化): 启动 chaincode 容器的过程 (在 lccc 中将 ChaincodeData 保存到 state 中, 然后部署 Chaincode 并执行 Init 方法)。

(20) Invoke (调用): 用于调用 chaincode 内的函数。Chaincode invoke 就是一个交易 proposal, 执行模块化的流程 (背书、共识、验证、提交)。invoke 的结构就是一个函数和



一个参数数组。

(21) Leading Peer (主导节点): 每一个 Member 在其订阅的 channel 上可以拥有多个 peer, 其中一个 peer 会作为 channel 的 leading peer 代表该 Member 与 ordering service 通信。ordering service 将 block 传递给 leading peer, 该 peer 再将此 block 分发给同一 member 下的其他 peer。

(22) Ledger (账本): Ledger 是 channel 的 chain 和由 channel 中每个 peer 维护的 world state。

(23) Member (成员): 拥有网络唯一根证书的合法独立实体。像 peer 节点和 client 这样的网络组件会链接到一个 Member。

(24) Membership Service Provider (MSP): MSP 是为 client 和 peer 提供证书的系统抽象组件。client 用证书来认证他们的交易; peer 用证书认证其交易背书。该接口与系统的交易处理组件密切相关, 旨在使已定义的成员身份服务组件以这种方式顺利插入而不会修改系统的交易处理组件的核心。

(25) Membership Services (成员服务): 成员服务在许可的区块链网络上认证、授权和管理身份。在 peer 和 order 中运行的成员服务的代码都会认证和授权区块链操作。它是基于 PKI 的 MSP 实现。

fabric-ca 组件实现了成员服务以管理身份, 处理 ECert 和 TCert 的颁发和撤销。

ECert 是长期的身份凭证; TCert 是短期的身份凭证, 是匿名和不可链接的。

(26) Ordering Service (排序服务或共识服务): 将交易排序放入 block 的节点的集合。ordering service 独立于 peer 流程之外, 并以先到先得的方式为网络上所有的 channel 提供交易排序服务。ordering service 支持可插拔实现, 目前默认实现了 SOLO 和 Kafka。ordering service 是整个网络的公用 binding, 包含与每个 Member 相关的加密材料。

(27) Peer (节点): 一个网络实体, 维护 ledger 并运行 Chaincode 容器来对 ledger 执行 read-write 操作。peer 由 Member 拥有和维护。

(28) Policy (策略): 有背书策略、校验策略、区块提交策略、Chaincode 管理策略和网络/通道管理策略。

(29) Proposal (提案): 一种针对 channel 中某 peer 的背书请求。每个 proposal 要么是 Chaincode instantiate, 要么是 Chaincode invoke。

(30) Query (查询): 针对 current state 中某个 key 的 value 的查询请求。

(31) Software Development Kit (SDK): SDK 为开发人员提供了一个结构化的库环境, 用于编写和测试链码应用程序。SDK 完全可以通过标准接口实现配置和扩展, 像签名的加密算法、日志框架和 state 存储这样的组件都可以轻松实现替换。SDK API 使用 gRPC 进行交易处理, 成员服务、节点遍历以及事件处理都是据此与 Fabric 通信。目前 SDK 支持 Node.js、Java 和 Python。

(32) State Database (stateDB): 为了从 Chaincode 中进行高效读写, Current state 数据



存储在 stateDB 中，包括 levelDB 和 couchDB。

(33) System Chain (系统链)：包含在系统级定义网络的配置区块中。系统链存在于 ordering service 中，与 channel 类似，具有包含以下信息的初始配置：MSP 信息、策略和信息配置。整个网络的任何变化（例如新的 Org 加入或者添加新的 Ordering 节点）都将导致新的配置区块被添加到系统链。

系统链可看作一个 channel 或一组 channel 的公用 binding。例如，金融机构的集合可以形成一个财团（以 system chain 表示），然后根据其相同或不同的业务创建 channel。

(34) Transaction (交易)：Chaincode 的 invoke 或 instantiate 操作。Invoke 是从 ledger 中请求 read-write set；Instantiate 是请求在 peer 上启动 Chaincode 容器。





## 作者简介

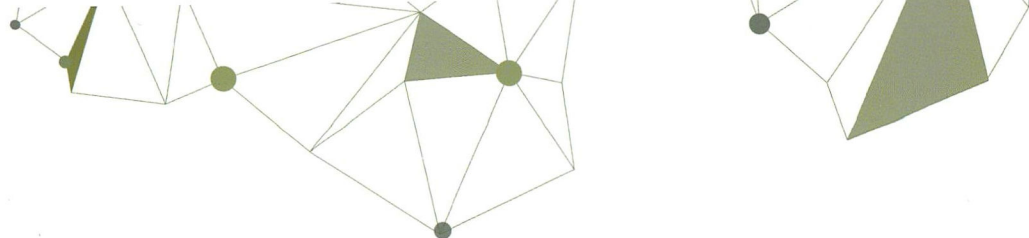
**蔡亮** 博士，副教授，浙江大学软件学院副院长、浙江大学区块链研究中心常务副主任、中国计算机学会区块链专业委员会副主任、中国可信区块链联盟副理事长，浙江省重大科技专项专家。

主要从事区块链、云计算、网络安全、可信计算和金融业务处理的研究，在国家级核心期刊和国际会议上发表了数十篇论文。参与了多项国家级或省部级科研项目，如国防军工预研基金项目、国家创新基金项目、863 项目等。获得教育部科技进步一等奖、浙江省科技进步一等奖和三等奖。

**梁秀波** 博士，浙江大学软件学院副研究员、浙江大学区块链研究中心主任助理、杭州趣链科技有限公司副总经理。

主要从事区块链、智能信息处理、金融信息技术和移动互联网等方面的研究与开发工作，曾赴法国进行为期一年的访问研究。主持或参与国家级和省部级科研项目十余项，主持企事业单位委托项目二十余项。已发表论文十余篇，已申请区块链方面的发明专利三十余项。

**宣章炯** 硕士，趣链科技高级架构师，曾就职于网易杭州研究院、阿里巴巴-蚂蚁金服事业群，有丰富的大型金融区块链项目开发经验。目前从事 Hyperledger Fabric 开源项目的研究工作，并为其贡献源代码，对其有较深理解，Hyperledger TWG-China 大中华区技术工作组成员，负责社区发展与创新工作及翻译工作，杭州地区 Meetup 的组织者，担任过 Meetup 的讲师。



区块链通过技术层面的信任机制大大降低了为保证多方信任而依赖中介等传统非技术手段所付出的成本，显著提升了业务效率。未来数字化社会中的各行各业都将受到该技术的深刻影响，这使得区块链技术成为当下最炙手可热的新兴 IT 技术之一。Hyperledger Fabric 是面向商业区块链应用开发的企业级区块链底层支撑平台，源代码开源，吸引了世界范围内的众多开发者竞相学习和贡献代码，是了解区块链理念和探知区块链技术的良好研习平台。本书以 Hyperledger Fabric 为研究对象，全面剖析了其架构设计理念和具体源码实现。通过本书的学习，你将具备从事区块链底层技术开发的基础能力，以及基于 Hyperledger Fabric 快速上手开发区块链应用项目的能力。

### 通过阅读本书，你将：

- 能够搭建 Hyperledger Fabric 开发环境
- 了解 Hyperledger Fabric 项目的架构设计
- 熟悉 Hyperledger Fabric 重要模块、节点和功能的源码实现
- 深入学习网络通信、加密算法和共识机制等区块链基础技术
- 熟练掌握 Fabric 智能合约（Chaincode）的设计、实现和使用
- 实现区块链项目案例从学习、开发到上线部署的全流程
- 具备区块链底层平台和上层应用开发的基本素质
- 了解和参与 Hyperledger 开源社区的各种活动

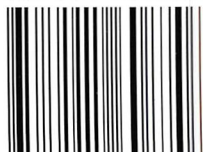


投稿热线：(010) 88379604  
客服热线：(010) 88379426 88361066  
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com  
网上购书：www.china-pub.com  
数字阅读：www.hzmedia.com.cn

上架指导：计算机/区块链

ISBN 978-7-111-60870-7



9 787111 608707

定价：89.00元